

普通高等教育“十一五”国家级规划教材



21世纪大学本科计算机专业系列教材

王晓东 编著

算法设计与分析习题解答（第4版）

http://www.tup.com.cn

- 国家精品课程配套教材
- 根据教育部“高等学校计算机科学与技术专业规范”组织编写
- 与美国 ACM 和 IEEE CS *Computing Curricula* 最新进展同步

清华大学出版社



普通高等教育“十一五”国家级规划教材
21 世纪大学本科计算机专业系列教材
国家精品课程配套教材

算法设计与分析习题解答

(第 4 版)

王晓东　编著

清华大学出版社
北京

内 容 简 介

本书是与清华大学出版社出版的普通高等教育"十一五"国家级规划教材《算法设计与分析(第4版)》(主教材)配套的辅助教材,对《算法设计与分析(第4版)》一书中的全部习题做了详尽的解答。本书的内容是对《算法设计与分析(第4版)》的较深入的扩展,将许多在主教材中无法讲述的、较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力,本书将主教材中的许多习题改造成算法实现题,要求学生不仅能设计解决具体问题的算法,而且能够上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设,进行了教材的立体化开发,包括主教材、辅助教材、实验与设计、电子课件和教学网站建设。

本书内容丰富,观点新颖,理论联系实际,不仅可以用作高等学校计算机类专业本科生和研究生学习计算机算法设计与分析的辅助教材,而且也适合广大工程技术人员和其他自学读者学习参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

算法设计与分析习题解答/王晓东编著.—4版.—北京:清华大学出版社,2018
(21世纪大学本科计算机专业系列教材)
ISBN 978-7-302-51106-9

Ⅰ.①算… Ⅱ.①王… Ⅲ.①电子计算机—算法设计—高等学校—题解 ②电子计算机—算法分析—高等学校—题解 Ⅳ.①TP301.6-44

中国版本图书馆CIP数据核字(2018)第195619号

责任编辑:张瑞庆
封面设计:何凤霞
责任校对:焦丽丽
责任印制:丛怀宇

出版发行:清华大学出版社
　　网　址:http://www.tup.com.cn, http://www.wqbook.com
　　地　址:北京清华大学学研大厦A座　　**邮　编**:100084
　　社 总 机:010-62770175　　**邮　购**:010-62786544
　　投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn
　　质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn
　　课件下载:http://www.tup.com.cn,010-62795954
印 刷 者:北京富博印刷有限公司
装 订 者:北京市密云县京文制本装订厂
经　销:全国新华书店
开　本:185mm×260mm　　**印　张**:25.5　　**字　数**:616千字
版　次:2003年1月第1版　2018年11月第4版　　**印　次**:2018年11月第1次印刷
定　价:59.00元

产品编号:079853-01

21世纪大学本科计算机专业系列教材编委会

主　　任：李晓明

副 主 任：蒋宗礼　卢先和

委　　员：（按姓氏笔画为序）

马华东　马殿富　王志英　王晓东　宁　洪

刘　辰　孙茂松　李仁发　李文新　杨　波

吴朝晖　何炎祥　宋方敏　张　莉　金　海

周兴社　孟祥旭　袁晓洁　钱乐秋　黄国兴

曾　明　廖明宏

秘　　书：张瑞庆

前　言

FOREWORD

计算机科学体现了创造性思维活动，其教育必须面向设计。"算法设计与分析"正是一门面向设计，并且处于计算学科核心地位的教育课程。通过对计算机算法系统的学习与研究，可以理解和掌握算法设计的主要方法，培养对算法的计算复杂性进行正确分析的能力，为独立地设计算法以及对给定算法进行复杂性分析奠定坚实的理论基础。学习和掌握计算机算法，对从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者是非常重要和必不可少的。

《算法设计与分析(第 4 版)》结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，在内容选材、深度把握、系统性和可用性方面进行了精心设计，力图适合高等学校本科生和研究生教学对学时数和知识结构的要求。

根据作者多年的教学经验，"算法设计与分析"课程教学有以下特点，使得许多学生感到学习相当困难。

(1) "算法设计与分析"课程教学包括的知识点多，内容十分丰富，学习量大。

(2) 课程内容理论性很强，对学生的抽象思维能力和逻辑推理能力要求较高。

(3) 课程内容还有很强的实践性，要求学生能够灵活地运用所学的算法设计策略解决实际问题。

课后习题能在很大程度上解决上面所说的困难。主教材《算法设计与分析(第 4 版)》所配备的习题正是为此目的而设计的。主教材出版后，许多读者纷纷要求给出习题的解答或提示。为了让使用《算法设计与分析(第 4 版)》作为教材的教师和学生在广度和深度各个层面，更加深刻地理解理论、抽象和设计这 3 个过程以及重复出现的 12 个基本概念，作者根据多年的教学经验编写了这本配套的辅助教材，旨在让使用《算法设计与分析(第 4 版)》的教师更容易教，学生更容易学。为了便于对照阅读，本书的章序与《算法设计与分析(第 4 版)》的章序保持一致，并且一一对应。

本书的内容是对《算法设计与分析(第 4 版)》的较深入的扩展，将许多在主教材中无法讲述的、较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力，本书将主教材中的许多习题改造成算法实现题，要求学生不仅能设计解决具体问题的算法，而且能够上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设，建设了"算法设计与分析"课程教学网站。

本书在网站上所附的电子文件中有各章算法实现题的题目、测试数据和答案,可以从清华大学出版社网站 www.tup.com.cn 免费下载。该电子文件共有 12 个子目录:ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8,ch9,ch10,ch11,exam。ch1,ch2,…,ch11 包括相应各章的算法实现题。每个算法实现题有两个子目录,其中的 test 子目录中是测试数据,answer 子目录中是相应的答案。在子目录 exam 中,midexam1 和 midexam2 是两套期中试卷,finalexam1 和 finalexam2 是两套期终试卷。

在本书的编写过程中,福州大学"211 工程"计算机与信息工程重点学科实验室为本书的写作提供了优良的设备与工作环境。清华大学出版社负责本书编辑出版工作的全体人员为本书的出版付出了大量辛勤劳动,他们认真细致、一丝不苟的工作作风保证了本书的出版质量。在此,谨向每一位曾经关心和支持本书编写和出版工作的各方面人士表示衷心的谢意!

作　者

2018 年 6 月

目　录

CONTENTS

第 1 章 算法引论

习题 1-1　实际参数交换

说明下面的方法 swap 为什么无法交换实际参数的值。

```
public static void swap(int x, int y)
{
    int temp=x;
    x=y;
    y=temp;
}
```

分析与解答：

Java 中所有方法的参数均为值参数，执行调用后，实际参数的值不变。

习题 1-2　方法头签名

说明下面的两个方法头是否有不同的签名，为什么？

(1) public int fff(int i,int j, int k)

(2) public float fff(int i,int j, int k)

分析与解答：

有相同的签名(int,int,int)，具体分析略。

习题 1-3　数组排序判定

写一个通用方法用于判定给定数组是否已排好序。

分析与解答：

```
public static boolean IsOrdered(Comparable []a)
{
    int j=0,n=a.length;
    while(j<n-1 && a[j].compareTo(a[j+1])==0)j++;
    if(n<=1 || j==n-1)return true;
    else j=a[j].compareTo(a[j+1]);
    for (int i=1;i<n-1;i++)
        if (a[i].compareTo(a[i+1])*j<0) return false;
    return true;
}
```

习题 1-4　函数的渐近表达式

求下列函数的渐近表达式。

(1) $3n^2+10n$

(2) $n^2/10+2^n$

(3) $21+1/n$

(4) $\log n^3$[①]

(5) $10\log 3^n$

分析与解答:

(1) $3n^2+10n=O(n^2)$。

(2) $n^2/10+2^n=O(2^n)$。

(3) $21+1/n=O(1)$。

(4) $\log n^3=O(\log n)$。

(5) $10\log 3^n=O(n)$。

习题 1-5　$O(1)$和$O(2)$的区别

说明 $O(1)$ 和 $O(2)$ 的区别。

分析与解答:

根据符号 O 的定义易知$O(1)=O(2)$。用$O(1)$或$O(2)$表示同一个方法时,差别仅在于其中的常数因子。

习题 1-6　按渐近阶排列表达式

按照渐近阶从低到高的顺序排列以下表达式: $4n^2$, $\log n$, 3^n, $20n$, 2, $n^{2/3}$。又 $n!$ 应该排在哪一位?

分析与解答:

按渐近阶从低到高,方法排列顺序为: 2, $\log n$, $n^{2/3}$, $20n$, $4n^2$, 3^n, $n!$。

习题 1-7　算法效率

(1) 假设某算法在输入规模为 n 时的计算时间为 $T(n)=3\times 2^n$。在某台计算机上实现并完成该算法的时间为 t 秒。现有另一台计算机,其运行速度为第一台计算机的 64 倍,那么在这台新机器上用同一算法在 t 秒内能解输入规模为多大的问题?

(2) 若上述算法的计算时间改进为 $T(n)=n^2$,其余条件不变,则在新机器上用 t 秒时间能解输入规模为多大的问题?

(3) 若上述算法的计算时间进一步改进为 $T(n)=8$,其余条件不变,那么在新机器上用 t 秒时间能解输入规模为多大的问题?

分析与解答:

(1) 设新机器用同一算法在 t 秒内能解输入规模为 n_1 的问题。因此有 $t=3\times 2^n=3\times 2^{n_1}/64$,解得 $n_1=n+6$。

(2) $n_1^2=64n^2\Rightarrow n_1=8n$。

---

① 本书除特殊说明外,所有对数均是以 2 为底的对数。

(3) 由于 $T(n)$等于常数，因此算法可解任意规模的问题。

习题 1-8　硬件效率

硬件厂商 XYZ 公司宣称最新研制的微处理器运行速度为其竞争对手 ABC 公司同类产品的 100 倍。对于计算复杂性分别为 n, n^2, n^3 和 $n!$ 的各算法，若用 ABC 公司的计算机在 1 小时内能解输入规模为 n 的问题，那么用 XYZ 公司的计算机在 1 小时内分别能解输入规模为多大的问题？

分析与解答：

$n' = 100n$。

$n'^2 = 100n^2 \Rightarrow n' = 10n$。

$n'^3 = 100n^3 \Rightarrow n' = \sqrt[3]{100}\,n = 4.64n$。

$n'! = 100n! \Rightarrow n' < n + \log 100 = n + 6.64$。

习题 1-9　函数渐近阶

对于下列各组函数 $f(n)$ 和 $g(n)$，确定 $f(n) = O(g(n))$ 或 $f(n) = \Omega(g(n))$ 或 $f(n) = \theta(g(n))$，并简述理由。

(1) $f(n) = \log n^2$；$g(n) = \log n + 5$

(2) $f(n) = \log n^2$；$g(n) = \sqrt{n}$

(3) $f(n) = n$；$g(n) = \log^2 n$

(4) $f(n) = n\log n + n$；$g(n) = \log n$

(5) $f(n) = 10$；$g(n) = \log 10$

(6) $f(n) = \log^2 n$；$g(n) = \log n$

(7) $f(n) = 2^n$；$g(n) = 100n^2$

(8) $f(n) = 2^n$；$g(n) = 3^n$

分析与解答：

(1) $\log n^2 = \theta(\log n + 5)$。

(2) $\log n^2 = O(\sqrt{n})$。

(3) $n = \Omega(\log^2 n)$。

(4) $n\log n + n = \Omega(\log n)$。

(5) $10 = \theta(\log 10)$。

(6) $\log^2 n = \Omega(\log n)$。

(7) $2^n = \Omega(100n^2)$。

(8) $2^n = O(3^n)$。

习题 1-10　$n!$的阶

证明：$n! = O(n^n)$。

分析与解答：

Stirling's approximation：$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right)$

$$\lim_{n\to\infty} n!/n^n = \frac{\sqrt{2\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{e^n} = 0$$

习题 1-11　平均情况下的计算时间复杂性

证明：如果一个算法在平均情况下的计算时间复杂性为 $\theta(f(n))$，则该算法在最坏情况下所需的计算时间为 $\Omega(f(n))$。

分析与解答：

$$\begin{aligned} T_{\text{avg}}(N) &= \sum_{I \in D_N} P(I)T(N,I) \\ &\leqslant \sum_{I \in D_N} P(I) \max_{I' \in D_N} T(N,I') \\ &= T(N,I^*) \sum_{I \in D_N} P(I) \\ &= T(N,I^*) = T_{\max}(N) \end{aligned}$$

因此，$T_{\max}(N) = \Omega(T_{\text{avg}}(N)) = \Omega(\theta(f(n))) = \Omega(f(n))$。

算法实现题 1-1　统计数字问题

★ 问题描述

一本书的页码从自然数 1 开始顺序编码直到自然数 n。书的页码按照通常的习惯编排，每个页码都不含多余的前导数字 0。例如，第 6 页用数字 6 表示，而不是 06 或 006 等。数字计数问题要求对给定书的总页码 n，计算出书的全部页码中分别用到多少次数字 0,1,2,…,9。

★ 算法设计

给定表示书的总页码的十进制整数 n(其中 $1 \leqslant n \leqslant 10^9$)。计算书的全部页码中分别用到多少次数字 0,1,2,…,9。

★ 数据输入

输入数据由文件名为 input. txt 的文本文件提供。

每个文件只有 1 行，给出表示书的总页码的整数 n。

★ 结果输出

将计算结果输出到文件 output. txt 中。输出文件共有 10 行，在第 k 行输出页码中用到数字 $k-1$ 的次数，$k=1,2,\cdots,10$。

输入文件示例	输出文件示例
input. txt	output. txt
11	1
	4
	1
	1
	1
	1
	1
	1
	1
	1

分析与解答：

考查由 0,1,2,…,9 组成的所有 n 位数。从 n 个 0 到 n 个 9 共有 10^n 个 n 位数。在这 10^n 个 n 位数中,0,1,2,…,9 每个数字使用次数相同,设为 $f(n)$。

$f(n)$ 满足如下递归式：

$$f(n)=\begin{cases}10f(n-1)+10^{n-1} & n>1\\ 1 & n=1\end{cases}$$

由此可知，$f(n)=n\,10^{n-1}$。

据此,可从高位向低位进行统计,再减去多余的 0 的个数即可。

算法实现题 1-2　字典序问题

★ 问题描述

在数据加密和数据压缩中常需要对特殊的字符串进行编码。给定的字母表 A 由 26 个小写英文字母组成 $A=\{a,b,\cdots,z\}$。该字母表产生的升序字符串指的是字符串中字母按照从左到右出现的次序与字母在字母表中出现的次序相同,且每个字符最多出现 1 次。例如,a,b,ab,bc,xyz 等字符串都是升序字符串。现在对字母表 A 产生的所有长度不超过 6 的升序字符串按照字典序排列并编码如下。

1	2	…	26	27	28	…
a	b	…	z	ab	ac	…

对于任意长度不超过 6 的升序字符串,迅速计算出它在上述字典中的编码。

★ 算法设计

对于给定的长度不超过 6 的升序字符串,计算出它在上述字典中的编码。

★ 数据输入

输入数据由文件名为 input.txt 的文本文件提供。

文件的第 1 行是 1 个正整数 k,表示接下来共有 k 行。

接下来的 k 行中,每行给出 1 个字符串。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件共有 k 行,每行对应 1 个字符串的编码。

输入文件示例	输出文件示例
input.txt	output.txt
2	1
a	2
b	

分析与解答：

考查一般情况下长度不超过 k 的升序字符串。

设以第 i 个字符打头的长度不超过 k 的升序字符串的个数为 $f(i,k)$，长度不超过 k 的升序字符串的总个数为 $g(k)$，则 $g(k)=\sum_{i=1}^{26}f(i,k)$。

易知，$f(i,1)=1$，$g(1)=\sum_{i=1}^{26}f(i,1)=26$。

$$f(i,2)=\sum_{j=i+1}^{26} f(j,1)=26-i,\quad g(2)=\sum_{i=1}^{26} f(i,2)=\sum_{i=1}^{26}(26-i)=325$$

一般情况下有

$$f(i,k)=\sum_{j=i+1}^{26} f(j,k-1),\quad g(k)=\sum_{i=1}^{26} f(i,k)=\sum_{i=1}^{26}\sum_{j=i+1}^{26} f(j,k-1)$$

据此可计算出每个升序字符串的编码。

算法实现题 1-3　最多约数问题

★ 问题描述

正整数 x 的约数是能整除 x 的正整数。正整数 x 的约数个数记为 div(x)。例如,1,2,5,10 都是正整数 10 的约数,且 div(10)=4。设 a 和 b 是 2 个正整数,$a \leqslant b$,找出 a 和 b 之间约数个数最多的数 x。

★ 算法设计

对于给定的 2 个正整数 $a \leqslant b$,计算 a 和 b 之间约数个数最多的数。

★ 数据输入

输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行有 2 个正整数 a 和 b。

★ 结果输出

若找到的 a 和 b 之间约数个数最多的数是 x,将 div(x)输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
1　36	9

分析与解答:

设正整数 x 的质因子分解为 $x=p_1^{N_1}p_2^{N_2}\cdots p_k^{N_k}$,则 $\text{div}(x)=(N_1+1)(N_2+1)\cdots(N_k+1)$。

搜索区间$[a,b]$中数的质因子分解。

primes 产生质数。

```
public static final int MAXP=31622;
public static final int PCOUNT=3401;
public static void primes()
{
    prim=new int [PCOUNT+1];
    boolean []get=new boolean [MAXP+1];
    for(int i=0;i<=PCOUNT;i++) prim[i]=0;
    for(int i=2;i<=MAXP;i++) get[i]=true;
    for(int i=2;i<=MAXP;i++)
        if(get[i]){
            int j=i+i;
            while(j<=MAXP){get[j]=false;j+=i;}
        }
    for(int ii=2,j=0;ii<=MAXP;ii++)
        if(get[ii]) prim[++j]=ii;
}
```

下面的search搜索最多约数。

```
public static void search(int from,int tot,int num,int low,int up)
{
    if(num>=l)
        if((tot>max) || ((tot==max)&&(num<numb))){max=tot;numb=num;}
    if((low==up)&&(low>num))search(from,tot * 2,num * low,1,1);
    for(int i=from;i<=PCOUNT;i++)
        if(prim[i]>up) return;
        else{
            int j=prim[i],x=low-1,y=up,n=num,t=tot,m=1;
            while(true){
                m++;t+=tot;x/=j;y/=j;
                if(x==y) break;
                n *=j;
                search(i+1,t,n,x+1,y);
            }
            m=1<<m;
            if(tot<max/m) return;
        }
}
```

实现算法的主方法如下：

```
public static void main(String [] args)
{
    ReadStreams keyboard=new ReadStreams();
    primes();
    l=keyboard.readInt();
    u=keyboard.readInt();
    if((l==1)&&(u==1)){max=1;numb=1;}
    else{max=2;numb=1;search(1,1,1,l,u);}
    System.out.println(max);
}
```

算法实现题1-4 金币阵列问题

★问题描述

有$m \times n(m \leqslant 100, n \leqslant 100)$个金币在桌面上排成一个$m$行$n$列的金币阵列。每一枚金币或正面朝上或背面朝上。用数字表示金币状态，0表示金币正面朝上，1表示背面朝上。

金币阵列游戏的规则如下：

(1) 每次可将任一行金币翻过来放在原来的位置上。

(2) 每次可任选2列，交换这2列金币的位置。

★算法设计

给定金币阵列的初始状态和目标状态，计算按金币游戏规则，将金币阵列从初始状态变换到目标状态所需的最少变换次数。

★数据输入

由文件input.txt给出输入数据。文件中有多组数据。文件的第1行有1个正整数k，

表示有k组数据。每组数据的第1行有2个正整数m和n。以下的m行是金币阵列的初始状态,每行有n个数字,表示该行金币的状态,0表示金币正面朝上,1表示背面朝上。接着的m行是金币阵列的目标状态。

★ **结果输出**

将计算出的最少变换次数按照输入数据的次序输出到文件output.txt。相应数据无解时输出−1。

输入文件示例	输出文件示例
input.txt	output.txt
2	2
4 3	−1
1 0 1	
0 0 0	
1 1 0	
1 0 1	
1 0 1	
1 1 1	
0 1 1	
1 0 1	
4 3	
1 0 1	
0 0 0	
1 0 0	
1 1 1	
1 1 0	
1 1 1	
0 1 1	
1 0 1	

分析与解答:

枚举初始状态每一列变换为目标状态第1列的情况。算法描述如下:

```
public static final int Size=100;
public static int [][]b0;
public static int [][]b1;
public static int [][]b;
public static int k,n,m,count,best;
public static boolean found;

public static void main(String [ ] args)
{
    ReadStreams keyboard = new ReadStreams();
    b=new int [Size+1][Size+1];
```

```
        b0=new int [Size+1][Size+1];
        b1=new int [Size+1][Size+1];
        k=keyboard.readInt();
        for(int i=1;i<=k;i++)
        {
            n=keyboard.readInt();
            m=keyboard.readInt();
            for(int x=1;x<=n;x++)
                for(int y=1;y<=m;y++)
                    b0[x][y]=keyboard.readInt();
            for(int x=1;x<=n;x++)
                for(int y=1;y<=m;y++)
                    b1[x][y]=keyboard.readInt();
            acpy(b,b1);best=m+n+1;
            for(int j=1;j<=m;j++)
            {
                acpy(b1,b);count=0;trans2(1,j);
                for(int p=1;p<=n;p++)
                    if(b0[p][1]!=b1[p][1])trans1(p);
                for(int p=1;p<=m;p++)
                {
                    found=false;
                    for(int q=p;q<=m;q++)
                        if(same(p,q)){trans2(p,q);found=true;break;}
                    if(!found) break;
                }
                if(found && count<best)best=count;
            }
            if(best<m+n+1) System.out.println(best);
            else System.out.println(-1);
        }
    }
```

其中,trans1 模拟行翻转变换;trans2 模拟列交换变换。

```
    public static void trans1(int x)
    {
        for(int i=1;i<=m;i++)b1[x][i]=b1[x][i]^1;
        count++;
    }

    public static void trans2(int x,int y)
    {
        for(int i=1;i<=n;i++) MyMath.swap(b1,i,x,i,y);
        if(x!=y)count++;
    }
```

```
public static boolean same(int x,int y)
{
    for(int i=1;i<=n;i++) if(b0[i][x]!=b1[i][y]) return false;
    return true;
}
public static void acpy(int [][]a,int [][]b)
{
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            a[i][j]=b[i][j];
}
```

算法实现题 1-5　最大间隙问题

★ 问题描述

最大间隙问题：给定 n 个实数 $x_1,x_2,\cdots,x_n$，求这 n 个数在实轴上相邻 2 个数之间的最大差值。假设对任何实数的下取整方法耗时为 $O(1)$，设计解最大间隙问题的线性时间算法。

★ 算法设计

对于给定的 n 个实数 $x_1,x_2,\cdots,x_n$，计算它们的最大间隙。

★ 数据输入

输入数据由文件名为 input. txt 的文本文件提供。文件的第 1 行有 1 个正整数 n。第 2 行中有 n 个实数 $x_1,x_2,\cdots,x_n$。

★ 结果输出

将找到的最大间隙输出到文件 output. txt 中。

输入文件示例	输出文件示例
input. txt	output. txt
5	3.2
2.3 3.1 7.5 1.5 6.3	

分析与解答：

用鸽舍原理设计最大间隙问题的线性时间算法如下：

```
public static double maxgap(int n,double []x)
{
    double minx=x[mini(n,x)],maxx=x[maxi(n,x)];
    // 用 n-2 个等间距点分割区间[minx,maxx]
    // 产生 n-1 个桶,每个桶 i 中用 high[i]和 low[i]分别存储分配给桶 i 的数中的最大数和最小数
    int []count=new int[n+1];
    double []low=new double[n+1];
    double []high=new double[n+1];
    // 桶初始化
    for(int i=1;i<=n-1;i++){
        count[i]=0; low[i]=maxx; high[i]=minx;
    }
    // 将 n 个数置于 n-1 个桶中
```

```
        for(int i=1;i<=n;i++){
            int bucket=(int)((n-1)*(x[i]-minx)/(maxx-minx))+1;
            count[bucket]++;
            if(x[i]<low[bucket])low[bucket]=x[i];
            if(x[i]>high[bucket])high[bucket]=x[i];
        }
        // 此时,除了 maxx 和 minx 外的 n-2 个数被置于 n-1 个桶中
        // 由鸽舍原理即知,至少有一个桶是空的
        // 这意味着最大间隙不会出现在同一桶中的两个数之间
        // 对每一个桶作一次线性扫描即可找出最大间隙
        double tmp=0,left=high[1];
        for(int i=2;i<=n-1;i++)
            if(count[i]>0){
                double thisgap=low[i]-left;
                if(thisgap>tmp)tmp=thisgap;
                left=high[i];
            }
        return tmp;
    }
```

其中,mini 和 maxi 分别计算数组中最小元素和最大元素的下标。

```
    public static int mini(int n,double []x)
    {
        double tmp=x[1];
        int k=1;
        for(int i=1;i<=n;i++)
            if(x[i]<tmp){tmp=x[i];k=i;}
        return k;
    }

    public static int maxi(int n,double []x)
    {
        double tmp=x[1];
        int k=1;
        for(int i=1;i<=n;i++)
            if(x[i]>tmp){tmp=x[i];k=i;}
        return k;
    }
```

由于下取整方法耗时为 $O(1)$,故循环体内的运算耗时为 $O(1)$。因此,整个算法耗时为 $O(n)$,即算法 maxgap 是求最大间隙问题的线性时间算法。注意到在代数判定树计算模型下,$\Omega(n\log n)$ 是最大间隙问题的一个计算时间下界。这意味着在代数判定树的计算模型下,最大间隙问题是不可能有线性时间算法的。在此题中假设下取整方法耗时为 $O(1)$,实际上这可以看作是在代数判定树计算模型中,将下取整运算作为基本运算增加到原有的基本运算集中,从而使代数判定树计算模型的计算能力得到增强。因此,可以在线性时间内解最大间隙问题。

第 2 章 递归与分治策略

习题 2-1　Hanoi 塔问题的非递归算法

证明 Hanoi 塔问题的递归算法与非递归算法实际上是一回事。

分析与解答：

Hanoi 塔问题的递归算法如下：

```
public static void hanoi(int n,int A,int B,int C)
{
    if(n>0)

        hanoi(n-1,A,C,B);
        move(n,A,B);
        hanoi(n-1,C,B,A);
    }
}
```

主教材中所述非递归算法的目的塔座不确定。当 n 为奇数时，目的塔座是 B，按顺时针方向移动；当 n 为偶数时，目的塔座为 C，按反时针方向移动。为确定起见，规定目的塔座为 B。Hanoi 塔问题的非递归算法可描述如下：

```
public static void hanoi(int n)
{
    int []top={0,0,0};
    int [][]tower=new int[n+1][3];
    int x,y,min=0;
    boolean b,bb;
    for(int i=0;i<=n;i++){tower[i][0]=n-i+1;tower[i][1]=n+1;tower[i][2]=n+1;}
    top[0]=n;b=odd(n);bb=true;
    while(top[1]<n){
        if(bb){
            x=min;
            if(b)y=(x+1)%3;else y=(x+2)%3;
            min=y;bb=false;
        }
```

```
        else{
            x=(min+1)%3;y=(min+2)%3;bb=true;
            if(tower[top[x]][x]>tower[top[y]][y]){int tmp=x;x=y;y=tmp;}
        }
        move(tower[top[x]][x],x+1,y+1);
        tower[top[y]+1][y]=tower[top[x]][x];
        top[x]--;top[y]++;
    }
}
```

下面用数学归纳法证明递归算法和非递归算法产生相同的移动序列。

当 $n=1$ 和 $n=2$ 时容易直接验证。设当 $k\leqslant n-1$ 时，递归算法和非递归算法产生完全相同的移动序列。考查 $k=n$ 的情形。

将移动分为顺时针移动(C)、逆时针移动(CC)和非最小圆盘塔座间的移动(O)。

当 n 为奇数时，顺时针非递归算法产生的移动序列为 C,O,C,O,…,C；逆时针非递归算法产生的移动序列为 CC,O,CC,O,…,CC。

当 n 为偶数时，顺时针非递归算法产生的移动序列为 CC,O,CC,O,…,CC；逆时针非递归算法产生的移动序列为 C,O,C,O,…,C。

(1) 当 n 为奇数时，顺时针递归算法 hanoi(n,A,B,C)产生的移动序列如下：

hanoi($n-1$,A,C,B)产生的移动序列,O,hanoi($n-1$,C,B,A) 产生的移动序列。

hanoi($n-1$,A,C,B)和 hanoi($n-1$,C,B,A)均为偶数圆盘逆时针移动问题。由数学归纳法知，产生的移动序列均为 C,O,C,O,…,C。因此，hanoi(n,A,B,C)产生的移动序列为 C,O,C,O,…,C。

(2) 当 n 为偶数时，顺时针递归算法 hanoi(n,A,B,C)产生的移动序列如下：

hanoi($n-1$,A,C,B)产生的移动序列,O,hanoi($n-1$,C,B,A) 产生的移动序列。

hanoi($n-1$,A,C,B)和 hanoi($n-1$,C,B,A)均为奇数圆盘逆时针移动问题。由数学归纳法知，产生的移动序列均为 CC,O,CC,O,…,CC。因此，hanoi(n,A,B,C)产生的移动序列为 CC,O,CC,O,…,CC。

当 n 为奇数和偶数时的逆时针递归算法也类似。

由数学归纳法即知，递归算法和非递归算法产生相同的移动序列。

习题 2-2　7 个二分搜索算法

下面的 7 个算法与本章中的二分搜索算法 binarySearch 略有不同。请判断这 7 个算法的正确性。如果算法不正确，请说明产生错误的原因。如果算法正确，请给出算法的正确性证明。

```
public static int binarySearch1(int [] a, int x, int n)
{
    int left=0; int right=n-1;
    while (left <= right)
    {
        int middle=(left + right)/2;
        if (x == a[middle]) return middle;
        if (x>a[middle]) left=middle;
```

```
            else right=middle;
        }
        return -1;
    }

    public static int binarySearch2(int [] a, int x, int n)
    {
        int left=0; int right=n-1;
        while (left < right-1)
        {
            int middle=(left + right)/2;
            if (x < a[middle]) right=middle;
            else left=middle;
        }
        if (x==a[left]) return left;
        else return -1;
    }

    public static int binarySearch3(int [] a, int x, int n)
    {
        int left=0; int right=n-1;
        while (left +1!= right)
        {
            int middle=(left + right)/2;
            if (x >=a[middle]) left=middle;
            else right=middle;
        }
        if (x==a[left]) return left;
        else return -1;
    }

    public static int binarySearch4(int[] a, int x, int n)
    {
        if (n>0&&x>=a[0])
        {
            int left=0; int right=n-1;
            while (left < right)
            {
                int middle=(left + right)/2;
                if (x < a[middle]) right=middle-1;
                else left=middle;
            }
            if (x==a[left]) return left;
        }
        return -1;
    }

    public static int binarySearch5(int[] a, int x, int n)
```

```
{
    if (n>0&&x>=a[0])
    {
        int left=0; int right=n-1;
        while (left < right)
        {
            int middle=(left + right+1)/2;
            if (x < a[middle]) right=middle-1;
            else left=middle;
        }
        if (x==a[left]) return left;
    }
    return -1;
}

public static int binarySearch6(int[] a, int x, int n)
{
    if (n>0&&x>=a[0])
    {
        int left=0; int right=n-1;
        while (left < right)
        {
            int middle=(left + right+1)/2;
            if (x < a[middle]) right=middle-1;
            else left=middle+1;
        }
        if (x==a[left]) return left;
    }
    return -1;
}

public static int binarySearch7(int[] a, int x, int n)
{
    if (n>0&&x>=a[0])
    {
        int left=0; int right=n-1;
        while (left < right)
        {
            int middle=(left + right+1)/2;
            if (x < a[middle]) right=middle;
            else left=middle;
        }
        if (x==a[left]) return left;
    }
    return  1;
}
```

分析与解答:

算法 binarySearch1 与主教材中的算法 binarySearch 相比,数组段左右游标 left 和 right 的调整不正确,导致陷入死循环。

算法 binarySearch2 与主教材中的算法 binarySearch 相比,数组段左右游标 left 和 right 的调整不正确,导致当 $x=a[n-1]$时返回错误。

算法 binarySearch3 与正确算法 binarySearch5 相比,数组段左右游标 left 和 right 的调整不正确,导致当 $x=a[n-1]$时返回错误。

算法 binarySearch4 与正确算法 binarySearch5 相比,数组段左右游标 left 和 right 的调整不正确,导致陷入死循环。

算法 binarySearch5 正确,且当数组中有重复元素时,返回满足条件的最右元素。

算法 binarySearch6 与正确算法 binarySearch5 相比,数组段左右游标 left 和 right 的调整不正确,导致当 $x=a[n-1]$时返回错误。

算法 binarySearch7 与正确算法 binarySearch5 相比,数组段左右游标 left 和 right 的调整不正确,导致当 $x=a[0]$时陷入死循环。

习题 2-3　改写二分搜索算法

设 $a[0:n-1]$是已排好序的数组。请改写二分搜索算法,使得当搜索元素 x 不在数组中时,返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j。当搜索元素在数组中时,i 和 j 相同,均为 x 在数组中的位置。

分析与解答:

如下改写二分搜索算法。

```
public static boolean binarySearch(int[]a, int x, int left,int right,int[] ind)
{
    int middle;
    while (left <= right)
    {
        middle=(left + right)/2;
        if (x == a[middle]) {ind[0]=ind[1]=middle;return true;}
        if (x>a[middle]) left=middle + 1;
        else right=middle-1;
    }
    ind[0]=right;ind[1]=left;
    return false;
}
```

返回的 ind[1]是小于 x 的最大元素位置,ind[0]是大于 x 的最小元素位置。

习题 2-4　大整数乘法的 $O(nm^{\log(3/2)})$算法

给定两个大整数 u 和 v,它们分别有 m 和 n 位数字,且 $m<n$。用通常的乘法求 uv 的值需要 $O(mn)$ 时间。可以将 u 和 v 均看作是有 n 位数字的大整数,用本章介绍的分治法,在 $O(n^{\log 3})$ 时间内计算 uv 的值。当 m 比 n 小得多时,用这种方法就显得效率不够高。试设计一个算法,在上述情况下用 $O(nm^{\log(3/2)})$ 时间求出 uv 的值。

分析与解答:

当 m 比 n 小得多时,将 v 分成 n/m 段,每段 m 位。计算 uv 需要 n/m 次 m 位乘法运

算。每次 m 位乘法可以用主教材中的分治法计算，耗时为 $O(m^{\log 3})$。因此，算法所需的计算时间为 $O((n/m)m^{\log 3}) = O(nm^{\log(3/2)})$。

习题 2-5　5 次 $n/3$ 位整数的乘法

在用分治法求两个 n 位大整数 u 和 v 的乘积时，将 u 和 v 都分割为长度为 $n/3$ 位的 3 段。证明可以用 5 次 $n/3$ 位整数的乘法求得 uv 的值。按此思想设计一个求两个大整数乘积的分治算法，并分析算法的计算复杂性。（提示：n 位的大整数除以一个常数 k 可以在 $\theta(n)$ 时间内完成。符号 θ 所隐含的常数可能依赖于 k。）

分析与解答：

这个问题有更一般的解。将两个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段，可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。

事实上，设 $x = 2^{n/m}$，可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u = u_0 + u_1 x + \cdots + u_{m-1} x^{m-1}, \quad v = v_0 + v_1 x + \cdots + v_{m-1} x^{m-1}$$

$$w = uv = w_0 + w_1 x + w_2 x^2 + \cdots + w_{2m-2} x^{2m-2}$$

将 u、v 和 w 都视为关于变量 x 的多项式，并取 $2m-1$ 个不同的数 $x_1, x_2, \cdots, x_{2m-1}$ 代入多项式，可得

$$u(x_i) = u_0 + u_1 x_i + \cdots + u_{m-1} x_i^{m-1}, \quad v(x_i) = v_0 + v_1 x_i + \cdots + v_{m-1} x_i^{m-1}$$

$$w(x_i) = u(x_i)v(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + \cdots + w_{2m-2} x_i^{2m-2}$$

用矩阵形式表示为

$$\begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix}$$

设

$$\boldsymbol{B} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix},$$

则

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix} = \boldsymbol{B}^{-1} \begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix}$$

其中，$w(x_i) = u(x_i)v(x_i)$ 是两个 n/m 位数的乘法运算，共有 $2m-1$ 个乘法。其他均为加减法或数乘运算。

下面用 $m=3$ 的具体例子来说明。

设 $x = 2^{n/3}$，可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u = u_0 + u_1 x + u_2 x^2, \quad v = v_0 + v_1 x + v_2 x^2$$

$$w = uv = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$$

取5个数 $x_1,x_2,\cdots,x_5$ 为：$x_1=0$，$x_2=-2$，$x_3=2$，$x_4=-1$，$x_5=1$。代入多项式得

$$
\begin{aligned}
a &= w(x_1) = u_0v_0 \\
b &= w(x_2) = (u_0-2u_1+4u_2)(v_0-2v_1+4v_2) \\
c &= w(x_3) = (u_0+2u_1+4u_2)(v_0+2v_1+4v_2) \\
d &= w(x_4) = (u_0-u_1+u_2)(v_0-v_1+v_2) \\
e &= w(x_5) = (u_0+u_1+u_2)(v_0+v_1+v_2)
\end{aligned}
$$

$$
\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}
$$

$$
\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}
$$

解得

$$
\begin{aligned}
w_0 &= a \\
w_1 &= \frac{b-c-8(d-e)}{12} \\
w_2 &= \frac{-b-c+16(d+e)-30a}{24} \\
w_3 &= \frac{-b+c+2(d-e)}{12} \\
w_4 &= \frac{b+c-4(d+e)+6a}{24}
\end{aligned}
$$

$x_1,x_2,\cdots,x_5$ 的不同取法，可以得到不同的分解方法。

按此分解设计的求两个 n 位大整数乘积的分治算法需要5次 $n/3$ 位整数乘法。分割及合并步所需的加减法和数乘运算的时间为 $O(n)$。设 $T(n)$ 是算法所需的计算时间，则

$$
T(n) = \begin{cases} O(1) & n=1 \\ 5T(n/3)+O(n) & n>1 \end{cases}
$$

由此可得 $T(n)=O(n^{\log_3 5})$。

一般情况下，将两个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段，可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。由此设计出的求两个 n 位大整数乘积的分治算法需要 $2m-1$ 次 n/m 位整数乘法。分割及合并步所需的加减法和数乘运算的时间为 $O(n)$。因此，其计算时间 $T(n)$ 满足

$$
T(n) = \begin{cases} O(1) & n=1 \\ (2m-1)T(n/m)+O(n) & n>1 \end{cases}
$$

解此递归式可得 $T(n)=O(n^{\log_m(2m-1)})$。

习题 2-6 矩阵乘法

对任何非零偶数 n，总可以找到奇数 m 和正整数 k，使得 $n=m2^k$。为了求出两个 n 阶矩阵的乘积，可以把一个 n 阶矩阵分成 $m\times m$ 个子矩阵，每个子矩阵有 $2^k\times 2^k$ 个元素。当需要求 $2^k\times 2^k$ 的子矩阵的积时，使用 Strassen 算法。设计一个传统方法与 Strassen 算法相结合的矩阵相乘算法，对任何偶数 n，都可以求出两个 n 阶矩阵的乘积。并分析算法的计算时间复杂性。

分析与解答：

将 n 阶矩阵分块为 $m\times m$ 的矩阵。用传统方法求两个 m 阶矩阵的乘积需要计算 $O(m^3)$ 次两个 $2^k\times 2^k$ 矩阵的乘积。用 Strassen 矩阵乘法计算两个 $2^k\times 2^k$ 矩阵的乘积需要的计算时间为 $O(7^k)$，因此算法的计算时间为 $O(7^k m^3)$。

习题 2-7 多项式乘积

设 $P(x)=a_0+a_1x+\cdots+a_dx^d$ 是一个 d 次多项式。假设已有一算法能在 $O(i)$ 时间内计算一个 i 次多项式与一个 1 次多项式的乘积，以及一个算法能在 $O(i\log i)$ 时间内计算两个 i 次多项式的乘积。对于任意给定的 d 个整数 $n_1,n_2,\cdots,n_d$，用分治法设计一个有效算法，计算出满足 $P(n_1)=P(n_2)=\cdots=P(n_d)=0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$，并分析算法的效率。

分析与解答：

$$P(x)=\prod_{i=1}^{d}(x-n_i)=\prod_{i=1}^{\lfloor d/2\rfloor}(x-n_i)\prod_{i=\lfloor d/2\rfloor+1}^{d}(x-n_i)=P_1(x)P_2(x)$$

用分治法将 d 次多项式转化为两个 $d/2$ 次多项式乘积。

设用此算法计算 d 次多项式所需计算时间为 $T(d)$，则 $T(d)$ 满足如下递归式

$$T(d)=\begin{cases}O(1) & d=1\\ 2T(d/2)+O(d\log d) & d>1\end{cases}$$

解此递归式可得 $T(d)=O(d\log^2 d)$。

习题 2-8 不动点问题的 $O(\log n)$ 时间算法

设 n 个不同的整数排好序后存于 $T[1:n]$ 中。若存在下标 i，$1\leqslant i\leqslant n$，使得 $T[i]=i$，设计一个有效算法找到这个下标。要求算法在最坏情况下的计算时间为 $O(\log n)$。

分析与解答：

由于 n 个整数是不同的，因此对任意 $1\leqslant i\leqslant n-1$，有 $T[i]\leqslant T[i+1]-1$。

(1) 对于 $1<i\leqslant n-1$，当 $T[i]>i$ 时，对任意的 $1\leqslant j<n$，有 $T[j]\geqslant T[i]+j-i>i+j-i=j$。

(2) 对于 $1<i\leqslant n$，当 $T[i]<i$ 时，对任意的 $1\leqslant j<i$，有 $T[j]\leqslant T[i]-i+j<i-i+j=j$。

由(1)和(2)可知，用二分搜索算法可以在 $O(\log n)$ 时间内找到所需要的下标。

习题 2-9 主元素问题的线性时间算法

设 $T[0:n-1]$ 是 n 个元素的数组。对任一元素 x，设 $S(x)=\{i\mid T[i]=x\}$。当 $|S(x)|>n/2$ 时，称 x 为 T 的主元素。设计一个线性时间算法，确定 $T[0:n-1]$ 是否有一个主元素。

分析与解答：

如果 T 有一个主元素 x，则 x 是 T 的中位数。反之，如果 T 的中位数不是 T 的主元素，则 T 没有主元素。因此，用一个线性时间找中位数的算法可以在线性时间内判定 T 是否有一个主元素。

习题 2-10　无序集主元素问题的线性时间算法

若在习题 2-9 中，数组 T 中元素不存在序关系，只能测试任意两个元素是否相等，试设计一个有效算法，确定 T 是否有一主元素。算法的计算复杂性应为 $O(n\log n)$。更进一步，能找到一个线性时间算法吗?

分析与解答：

(1) 用分治法找 $T[1:n]$的主元素。设 x 是 $T[1:n]$的主元素。$S_x=\{i \mid T[i]=x\}$，则 $|S_x|>n/2$。将 $T[1:n]$分为两个大小相同的子数组 $T[1:n/2]$和 $T[n/2+1:n]$。易知 x 是 $T[1:n/2]$的主元素或 x 是 $T[n/2+1:n]$的主元素。换句话说，若这两个子数组均没有主元素，则 $T[1:n]$也没有主元素。为了判定一个候选元素 x 是否为 $T[i:j]$的主元素，只要对 $T[i:j]$进行一次线性扫描即可。在最坏情况下算法所需的计算时间 $T(n)$满足如下递归式

$$T(n)=\begin{cases}O(1) & n\leqslant 4\\ 2T(n/2)+O(n) & n>4\end{cases}$$

因此，$T(n)=O(n\log n)$。

(2) 用下面的算法设计策略可在 $O(n)$ 时间内找出 $T[1:n]$的主元素。

对 $1\leqslant i\leqslant n/2$，比较 $T[2*i-1]$与 $T[2*i]$。当 $T[2*i-1]=T[2*i]$时，将 $T[2*i]$存入另一数组 Q 中，否则什么也不做。

设经过这样的比较后 Q 中有 m 个元素，则 $m\leqslant n/2$。关于数组 Q 有如下结论：若 x 是 $T[1:n]$的主元素，则 x 是 Q 的主元素或 $T[n]$是 $T[1:n]$的主元素。

基于以上讨论，容易设计出在 $O(n)$ 时间内找出 $T[1:n]$的主元素的算法。

习题 2-11　$O(1)$空间子数组换位算法

设 $a[0:n-1]$是有 n 个元素的数组，k $(1\leqslant k\leqslant n-1)$是非负整数。试设计一个算法将子数组 $a[0:k-1]$与 $a[k:n-1]$换位。要求算法在最坏情况下耗时为 $O(n)$，且只用到 $O(1)$ 的辅助空间。

分析与解答：

算法 1： 循环换位算法

(1) 向前循环换位算法如下：

```
public static void forward(int []a, int n, int k)
{
    for(int i=0;i<k;i++)
    {
        int tmp=a[0];
        for(int j=1;j<n;j++)a[j-1]=a[j];
        a[n-1]=tmp;
    }
}
```

(2) 向后循环换位算法如下：

```
public static void backward(int [ ]a, int n, int k)
{
    for(int i=k;i<n;i++)
    {
        int tmp=a[n-1];
        for(int j=n-1;j>0;j--)a[j]=a[j-1];
        a[0]=tmp;
    }
}
```

(3) 选择较小的数组块进行循环算法如下：

```
public static void exch0(int[]a, int n, int k)
{// block exchange a[0:k-1] and a[k:n-1]
    if(k>n-k) backward(a,n,k);
    else forward(a,n,k);
}
```

在最坏情况下算法所需的元素移动次数为 $\min\{k,n-k\}*(n+1)$。算法仅用到一个辅助单元 tmp，因此，算法只用到 $O(1)$ 的辅助空间。当 $k=n/2$ 时，计算时间非线性。

算法 2：3 次反转算法

将数组块 $a[i:j]$ 反转的算法如下：

```
public static void reverse(int []a, int i, int j)
{
    while(i<j) {MyMath.swap(a, i, j);i++;j--;}
}
```

设 $a[0:k-1]$ 为 U，$a[k:n-1]$ 为 V，换位算法要求将 UV 变换为 VU。3 次反转算法先将 U 反转为 U^{-1}，再将 V 反转为 V^{-1}，最后将 $U^{-1}V^{-1}$ 反转为 VU。

```
public static void exch1(int []a, int n, int k)
{// block exchange a[0:k-1] and a[k:n-1]
    reverse(a,0,k-1);
    reverse(a,k,n-1);
    reverse(a,0,n-1);
}
```

3 次反转算法用了 $\lfloor k/2\rfloor+\lfloor(n-k)/2\rfloor+\lfloor n/2\rfloor\leqslant n$ 次数组单元交换运算。每个数组单元交换运算需要 3 次元素移动。因此，在最坏情况下，3 次反转算法用了 $3n$ 次元素移动。算法显然只用到 $O(1)$ 的辅助空间。

算法 3：排列循环算法

向后循环换位算法，实际上执行了数组元素的一个重新排列。因此，向后循环换位对应于 n 个元素的一个置换。这类置换具有如下的特殊性质。

循环置换分解定理：对于给定数组 $a[0:n-1]$ 向后循环换位 $n-k$ 位运算，可分解为恰好 $\gcd(k,n-k)$ 个循环置换，且 $0,\cdots,\gcd(k,n-k)-1$ 中每个数恰属于一个循环置换。其

中,$gcd(x,y)$表示整数x和y的最大公因数。

基于循环置换分解定理可设计下面的数组块换位算法。

```
public static void exch(int [ ]a, int n, int k)
{// block exchange a[0:k-1] and a[k:n-1]
    for(int i=0,cyc=gcd(k,n-k);i<cyc;i++)
    {
        int tmp=a[i];
        int p=i,j=(k+i)%n;
        while(j!=i){a[p]=a[j];p=j;j=(k+p)%n;}
        a[p]=tmp;
    }
}
```

上述算法总共移动元素$n+gcd(k,n-k)$次,算法显然只用到$O(1)$的辅助空间。

当换位数组块的长度相等时,算法更简单。例如,将数组块$a[b:b+l-1]$和$a[c:c+l-1]$换位的算法可表述如下。

```
public static void eqexch(int []a, int b, int c, int l)
{// equal size block exchange a[b:b+l-1] and a[c:c+l-1]
    for(int i=0;i<l;i++)MyMath.swap(a, b+i, c+i);
}
```

上述算法总共移动元素$3\times l$次。

习题 2-12　$O(1)$空间合并算法

设子数组$a[0:k-1]$和$a[k:n-1]$已排好序($0\leqslant k\leqslant n-1$)。试设计一个合并这两个子数组为排好序的数组$a[0:n-1]$的算法。要求算法在最坏情况下所用的计算时间为$O(n)$,且只用到$O(1)$的辅助空间。

分析与解答:

算法1:循环换位合并算法

1) 向右循环换位合并

向右循环换位合并算法首先用二分搜索算法在数组段$a[k:n-1]$中搜索$a[0]$的插入位置,即找到位置p使得$a[p]<a[0]\leqslant a[p+1]$。数组段$a[0:p]$向右循环换位$p-k+1$个位置,使$a[k-1]$移动到$a[p]$的位置。此时,原数组元素$a[0]$及其左边的所有元素均已排好序。对剩余的数组元素重复上述过程,直至只剩下一个数组段,此时,整个数组已排好序。

向右循环换位合并算法可描述如下:

```
public static void mergefor(int [ ]a, int k, int n)
{// Merge a[0:k-1] and c[k:n-1]
    int i=0,j=k;
    while(i<j && j<n){
        int p=binarySearch(a,a[i],j,n-1);
        shiftright(a,i,p,p-j+1);
        j=p+1;i+=p-j+2;
    }
```

```
}
```

算法 binarySearch(a, x, left, right)用于数组段 a[left:right]中搜索元素 x 的插入位置。

```
public static int binarySearch(int [ ]a, int x, int left,int right)
{
    int middle=0;
    while (left <= right){
        middle=(left + right)/2;
        if (x == a[middle]) return middle;
        if (x>a[middle]) left=middle + 1;
        else right=middle-1;
    }
    if (x>a[middle]) return middle;
    else return middle-1;
}
```

算法 shiftright(a, s, t, k)用于将数组段 $a[s:t]$中元素循环右移 k 个位置。

```
public static void shiftright(int []a, int s, int t, int k)
{
    for(int i=0;i<k;i++)
    {
        int tmp=a[t];
        for(int j=t;j>s;j--)a[j]=a[j-1];
        a[s]=tmp;
    }
}
```

上述算法中，数组段 $a[0:k-1]$中元素的移动次数不超过 k 次；数组段 $a[k:n-1]$中元素最多移动 1 次。因此，算法的元素移动总次数不超过 $k^2+(n-k)$次。算法的元素比较次数不超过 $k\log(n-k)$ 次。当 $k<\sqrt{n}$ 时，算法的计算时间为 $O(n)$。而当 $k=O(n)$ 时，算法的计算时间为 $O(n^2)$。由于数组段循环右移位算法只用了 $O(1)$ 的辅助空间，所以整个算法所用的辅助空间为 $O(1)$。

2）向左循环换位合并

类似地可以设计向左循环换位合并算法 mergeback。

算法 2：内部缓存算法

1）算法思想概述

为便于叙述，假定 n 是一个完全平方数，稍后讨论一般情况。图 2-1(a)是当 $n=25$，$k=12$ 时的一个示例。其中用大写字母表示数组元素的键值，下标表示相同键值出现的次序。

首先将待合并数组划分为 $\sqrt{n}$ 个数组块，每块的大小为 $\sqrt{n}$。将数组中最大的 $\sqrt{n}$ 个元素置于数组的最左端，作为算法的内部缓存，如图 2-1(b)所示。接下来用选择排序算法对除了内部缓存外的 $\sqrt{n}-1$ 个数组块排序，使数组块的最右端元素按非减序排列，每个数组块内部元素的相对次序保持不变。这个排序过程需要比较 $O(n)$ 次元素和移动 $O(n)$ 次元素。



(a) 给定的待合并数组

(b) 抽取内部缓存

(c) 数组块排序

图 2-1 数组块重排

用习题 2-11 中的算法 eqexch 可以保证上述排序过程只用 O(1) 的辅助空间。数组块排序后的结果如图 2-1(c)所示。

接下来要确定待合并的数组块序列。第 1 个序列是从数组块 2 开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块,如图 2-2(a)所示。



(a) 确定待合并序列

(b) 内部缓存合并过程

(c) 完成序列合并

图 2-2 合并两个序列

现在可以用内部缓存对两个序列进行合并。每次比较两个序列的最小元素,都将较小者与内部缓存的最左端元素交换位置。合并过程中内部缓存可能被分为两个不连续的段,如图 2-2(b)所示。当第 1 个序列的最后一个元素进入正确位置后,完成这一轮序列合并动作,此时内部缓存形成一个连续的块,且第 2 个序列中至少还有 1 个元素,如图 2-2(c)所示。

下一轮的序列合并是类似的。第 1 个序列是从内部缓存右端的下一个元素开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块,如图 2-3(a)所示。继续用内部缓存合并这两个序列,直至第 1 个序列完成合并,如图 2-3(b)所示。上述过程一直进行到只剩下一个序列时为止。此时只要将剩下的这个序列左移,内部缓存成为最后的一个数组块,如图 2-3(c)所示。此时内部缓存左边的所有元素均已排好序,且内部缓存中的元素是整个数组中最大的 $\sqrt{n}$ 个元素,用选择排序算法对内部缓存中的元素排序,完成整个合并过程,如

图 2-3(d)所示。

$A_1\ A_2\ A_3\ B_4\ B_5\ B_1\ B_2\ B_3\ C_1\ C_2\ D_1\ D_2$ | $I_1\ H_2\ H_3\ I_2\ J_1$ | $E_3\ G_2\ G_3$ | $E_1\ E_2\ F_1\ G_1\ H_1$

已合并序列 | 内部缓存 | 待合并序列1 | 待合并序列2

(a) 确定下一对待合并序列

$A_1\ A_2\ A_3\ B_4\ B_5\ B_1\ B_2\ B_3\ C_1\ C_2\ D_1\ D_2\ E_3\ E_1\ E_2\ F_1\ G_2\ G_3$ | $J_1\ I_1\ H_2\ H_3\ I_2$ | $G_1\ H_1$

内部缓存

(b) 完成序列合并

$A_1\ A_2\ A_3\ B_4\ B_5\ B_1\ B_2\ B_3\ C_1\ C_2\ D_1\ D_2\ E_3\ E_1\ E_2\ F_1\ G_2\ G_3\ G_1$ | $H_1\ H_2\ H_3\ I_2\ J_1\ I_1$

内部缓存

(c) 移位完成单序列合并

$A_1\ A_2\ A_3\ B_4\ B_5\ B_1\ B_2\ B_3\ C_1\ C_2\ D_1\ D_2\ E_3\ E_1\ E_2\ F_1\ G_2\ G_3\ G_1\ H_1\ H_2\ H_3\ I_2\ I_1\ J_1$

(d) 对内部缓存排序，完成整个合并

图 2-3　整个数组的合并过程

由于合并过程只用到内部缓存，上述整个合并过程只用 $O(1)$的辅助空间。数组块排序、序列合并以及内部缓存排序显然只需要 $O(n)$计算时间。因此，内部缓存算法需要 $O(n)$计算时间和 $O(1)$的辅助空间。

2）一般情况

在一般情况下，可以用 $O(\sqrt{n})$ 时间将问题转换为前面讨论的特殊情况。

当待合并的两个数组段中有一个数组段的长度小于$\sqrt{n}$时，可以用前面讨论过的循环换位合并算法，在 $O(n)$计算时间内，用 $O(1)$的辅助空间完成合并。

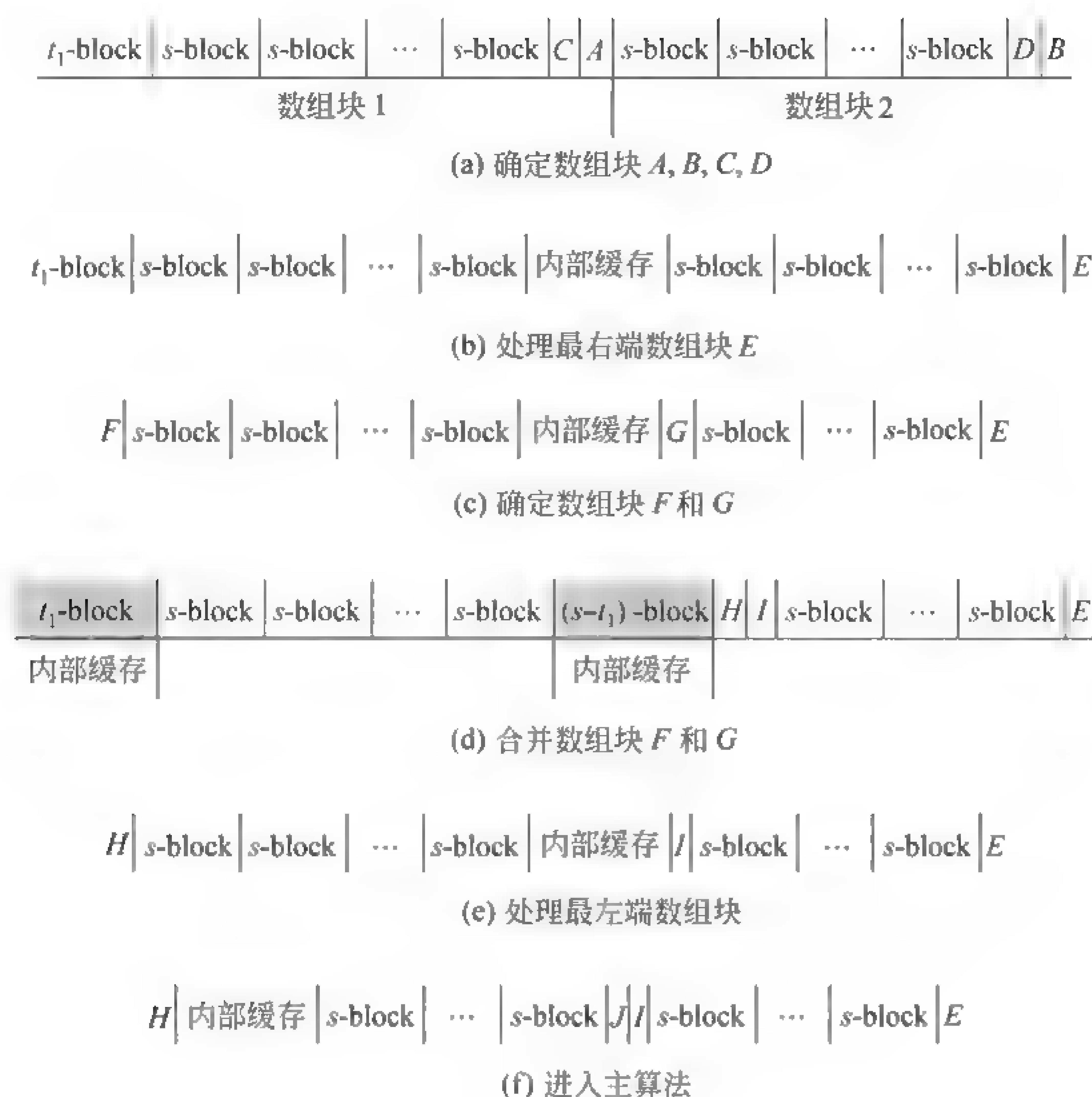
接下来的讨论中，设 $s=\lfloor\sqrt{n}\rfloor$，且 $\min\{k,n-k\}>\sqrt{n}$。数组中大小为 s 的块记为 s-block。由于 $(s+1)^2>n$，数组中 s-block 的个数不超过 $s+2$。

首先要找出数组中组成内部缓存的 s 个最大元素。一般情况下，这 s 个元素由数组中大小分别为 s_1 和 s_2 的两个部分 A 和 B 组成，$s_1+s_2=s$。紧挨着 A 的左边的 s_2 个元素记作 C。D 是紧挨着 B 的左边的数组块，其大小是使数组块 2 剩余元素为 s 的倍数的最小值。按此划分，可以将数组看作由 s-block 组成的数组块。除了第 1 块的大小 t_1 和最后一块的大小 t_2 外，每个数组块 s-block 的大小均为 s，而且 $0<t_1\leqslant s$，$0<t_2<2s$，如图 2-4(a)所示。

接下来交换数组块 C 和 B，使数组块 B 和 A 构成内部缓存。然后用内部缓存合并数组块 D 和 C，构成排好序的数组块 E，如图 2-4(b)所示。

当 $t_1<s$ 时，需要对最左端的 t_1-block 进行特殊处理。设 F 是这个特殊的数组块，G 是紧挨着内部缓存的 s-block，如图 2-4(c)所示。用内部缓存的最右 t_1 个单元将 F 和 G 合并得到 H 和 I，如图 2-4(d)所示。将 H 与最左端的内部缓存 t_1-block 交换，使 H 已在最终位置，且内部缓存连成一体，如图 2-4(e)所示。最后，将内部缓存与第 1 个 s-block 换位，转化为前面讨论过的规则情形，如图 2-4(f)所示。

以上这些处理只需要 $O(s)=O(\sqrt{n})$ 的计算时间和 $O(1)$的辅助空间。



(a) 确定数组块 A, B, C, D

(b) 处理最右端数组块 E

(c) 确定数组块 F 和 G

(d) 合并数组块 F 和 G

(e) 处理最左端数组块

(f) 进入主算法

图 2-4 对一般情况的处理

3) 算法实现

上面讨论的算法可以分为两个阶段和4个子任务,分别实现如下:

当 $k<\sqrt{n}$ 或 $n-k<\sqrt{n}$ 时,可以用前面讨论过的循环换位合并算法 mergefor 和 mergeback 在 $O(n)$ 计算时间内,用 $O(1)$ 的辅助空间完成合并。

```
public static void merge(int []a, int k, int n)
{// Merge a[0:k-1] and c[k:n-1]
    int s=(int)Math.sqrt(n);
    if(k<s){mergefor(a,k,n);return;}
    if(n-k<s){mergeback(a,k,n);return;}
    int j=task1(a,k,n,s);
    eqexch(a,k-s,j,n-j);
    bfs=k-s;bft=k-1;
    int ds=j-(j-k)%s,dt=j-1;
    task2(a,n,ds,dt);
    task3(a,k,n,s);
    maintask(a,k,n,s,ds);
}
```

算法的第1阶段是初始化阶段,分别由 task1、task2 和 task3 子任务来完成。第2阶段进入主算法,由子任务 maintask 来完成。下面分别讨论。

算法 task1 完成抽取内部缓存的任务，相应于图 2 4(a)，其返回值是数组块 B 的最左元素的下标。

```
public static int task1(int [ ]a, int k, int n, int s)
{
    int i=k-1,j=n-1;
    for(int t=0;t<s;t++)
    {
        if(a[i]<a[j])j ;
        else i--;
    }
    return j+1;
}
```

算法 task2 完成对最右数组块 E 的排序，相应于图 2-4(b)。

```
public static void task2(int []a,int n,int ds,int dt)
{
    if(ds>dt)return;
    selectionSort(a,ds,n-1);
}
```

算法 task3 完成对最左数组块的排序，相应于图 2-4(c)至图 2-4(f)。

```
public static void task3(int []a, int k, int n, int s)
{
    int t1=k%s;
    bfs=bft-t1+1;
    bfmerge(a,bft,0,t1-1,bft+1,bft+s);
    eqexch(a,0,bft-t1+1,t1);
    eqexch(a,t1,bft-s+1,s);
    bfs=t1;bft=bfs+s-1;
}
```

其中，变量 bfs 和 bft 是表示内部缓存块起始位置和终止位置的全局变量。算法中用到的数组块交换算法 eqexch 在习题 2-11 中已描述。用内部缓存进行合并的算法 bfmerge 表述如下：

```
public static void bfmerge(int [ ]a, int bt, int s1, int t1, int s2, int t2)
{// buffer Merge
    int ps1=s1;
    while(s1<=t1)
    {
        if(s2<=t2 && a[s1]>a[s2]){MyMath.swap(a, s2, bfs);s2++;bfs++;}
        else {MyMath.swap(a, s1, bfs);s1++;bfs++;}
        if(bfs==s2)bfs=ps1;
    }
}
```

算法 maintask 是主算法，完成最后的合并任务。

```
public static void maintask(int [ ]a, int k, int n, int s,int ds)
{
    sortblock(a,bfs+s,ds-1,s);
    while(bfs<n-s)
    {
        int s1=bfs+s,t1=s1+(ds-bfs-1)%s;
        while(t1<ds && a[t1]<=a[t1+1])t1+=s;
        if(t1>n-1)t1=n-1;
        int s2=t1+1,t2=t1+s;
        if(s2>ds)t2=n-1;
        bfmerge(a,bfs+s-1,s1,t1,s2,t2);
        if(s1>t1 && s2<=t2) {eqexch(a,bfs,s2,t2-s2-1);break;}
    }
    selectionSort(a,bfs,n-1);
}
```

其中,sortblock 是用选择排序对数组块进行排序的算法。

```
public static int maxblock(int []a,int left, int r,int s)
{
    int pos=1;
    for (int i=2; i <=r; i++)
        if (a[left+pos * s-1] < a[left+i * s-1]) pos=i;
    return pos;
}

public static void sortblock(int []a,int left,int right,int s)
{
    int m=(right-left+1)/s;
    for (int r=m; r >1; r--)
    {
        int j=maxblock(a,left,r,s);
        if(j<r) eqexch(a,left+(j-1) * s,left+(r-1) * s,s);
    }
}
```

习题 2-13 $\sqrt{n}$段合并排序算法

如果在合并排序算法的分割步中,将数组 $a[0:n-1]$划分为$\lfloor\sqrt{n}\rfloor$个子数组,每个子数组中有 $O(\sqrt{n})$ 个元素。然后递归地对分割后的子数组进行排序,最后将所得到的$\lfloor\sqrt{n}\rfloor$个排好序的子数组合并成所要求的排好序的数组 $a[0:n-1]$。设计一个实现上述策略的合并排序算法,并分析算法的计算复杂性。

分析与解答:

实现上述策略的合并排序算法如下:

```
public static void mergesort(int [ ]a, int left, int right)
{
```

```
if(left<right)
{
    int j=(int)Math.sqrt(right-left+1);
    if(j>1)
    {
        for(int i=0;i<j;i++) mergesort(a,left+i*j,left+(i+1)*j-1);
        mergesort(a,left+j*j,right);
    }
    mergeall(a,left,right);
}
}
```

其中,算法 mergeall 合并 $\sqrt{n}$ 个排好序的数组段。在最坏情况下,算法 mergeall 需要 $O(n\log n)$ 时间。因此,上述算法所需的计算时间 $T(n)$ 满足

$$T(n)=\begin{cases}O(1) & n\leqslant 1\\ \sqrt{n}T(\sqrt{n}) & n>1\end{cases}$$

此递归式的解为 $T(n)=O(n\log n)$。

习题 2-14　自然合并排序算法

对所给元素存储于数组中和存储于链表中两种情形,写出自然合并排序算法。

分析与解答:

对于所给元素存储于数组中的情形,自然合并排序算法如下:

```
public static void sort(int []a0,int m)
{
    a=a0;
    n=m;
    b=new int[n];
    naturalmergesort();
}

public static void naturalmergesort()
{
    while (!mergeruns(a, b) & !mergeruns(b, a));
}
```

由 mergeruns 实际完成自然合并排序算法。

```
public static boolean mergeruns(int []a, int []b)
{
    int i=0, k=0, x;
    boolean asc =true;
    while (i<n)
    {
        k=i;
        do x=a[i++]; while (i<n && x<=a[i]);
        while (i<n && x>=a[i]) x=a[i++];
```

```
        merge (a, b, k, i  1, asc);
        asc=!asc;
    }
    return k==0;
}
public static void merge(int []a, int []b, int lo, int hi, boolean asc)
{
    int k=asc ? lo : hi;
    int c=asc ? 1 :-1;
    int i=lo, j=hi;
    while (i<=j)
    {
        if (a[i]<=a[j]) b[k]=a[i++];
        else b[k]=a[j--];
        k+=c;
    }
}
```

对于所给元素存储于链表中的情形,自顶向下自然合并排序算法如下:
链表结点为

```
public class Node
{
    int item; Node next;
    Node(){item=0;next=null;}
}
static Node mergesort(Node c)
{
    if (c == null || c.next == null) return c;
    Node a=c, b=c.next;
    while ((b != null) && (b.next != null))
    { c=c.next; b=(b.next).next; }
    b=c.next; c.next=null;
    return merge(mergesort(a), mergesort(b));
}
```

算法 merge 实现已排序链表的合并。

```
static Node merge(Node a, Node b)
{
    Node c,head;
    c=head=new Node();
    while ((a != null) && (b != null))
        if (a.item<b.item){ c.next=a; c=a; a=a.next; }
        else{ c.next=b; c=b; b=b.next; }
    c.next=(a == null) ? b : a;
```

```
    return head.next;
}
```

自底向上自然合并排序算法如下：

```
static Node mergesort(Node t)
{
    LinkedQueue Q=new LinkedQueue();
    if (t == null || t.next == null) return t;
    for (Node u=null; t != null; t=u)
    { u=t.next; t.next=null; Q.put(t); }
    t=(Node)Q.remove();
    while (!Q.isEmpty())
    { Q.put(t); t=merge((Node)Q.remove(), (Node)Q.remove()); }
    return t;
}
```

习题 2-15　最大值和最小值问题的最优算法

给定数组 $a[0:n-1]$，试设计一个算法，在最坏情况下用 $\lceil 3n/2-2 \rceil$ 次比较找出 $a[0:n-1]$ 中元素的最大值和最小值。

分析与解答：

设所给的 n 个实数为 $x[i]$，$i=1\sim n$。

(1) 当 n 为偶数时，设 $n=2k$，首先，作 k 次比较 $x[i]:x[k+i]$，$1\leqslant i\leqslant k$。当 $x[k+i]>x[i]$时，交换它们的位置。这样，经 k 次比较后有 $x[i]\geqslant x[k+i]$，$1\leqslant i\leqslant k$。

然后，用 $k-1$ 次比较找出 $x[1:k]$中最大者，再用 $k-1$ 次比较找出 $x[k+1:n]$中最小者。找出的这两个数即为所要求的最大值与最小值。所用的比较次数为 $k+2k-2=3k-2=2n-\lfloor n/2 \rfloor-2$。

(2) 当 n 为奇数时，设 $n=2k+1$。首先，用对 n 为偶数时的方法找出 $x[1:n-1]$中的最大值与最小值。然后，再将 $x[n]$与找出的最大值与最小值作两次比较，即可确定所要的最大值与最小值。在这种情况下，所用的比较次数为 $3k-2+2=3k$，而 $2n-\lfloor n/2 \rfloor-2=4k+2-k-2=3k$。

结合(1)和(2)即知，所述算法需要的比较次数为 $2n-\lfloor n/2 \rfloor-2$。

由对手论证方法已证明该问题的计算时间下界(比较次数)为 $2n-\lfloor n/2 \rfloor-2$。因此，所述算法是求 n 个数的最大值与最小值的最优算法。

习题 2-16　最大值和次大值问题的最优算法

给定数组 $a[0:n-1]$，试设计一个算法，在最坏情况下用 $n+\lceil \log n \rceil-2$ 次比较找出 $a[0:n-1]$中元素的最大值和次大值。

分析与解答：

首先，将 $x[1:n]$分为两组：$x[1:k]$和 $x[k+1:n]$，其中 $k=n/2$。然后，作 k 次比较 $x[i]:x[k+i]$，$1\leqslant i\leqslant k$。当 $x[i]>x[k+i]$时，交换它们的位置。这样经过 k 次比较后有 $x[i]\leqslant x[k+i]$，$1\leqslant i\leqslant k$。递归地在 $x[k+1:n]$中找出其中的最大数 $x[p]$和次大数 $x[q]$。容易看出，$x[p]$即为 $x[1:n]$中的最大数。而 $x[1:n]$中的次大数只能是 $x[q]$或 $x[p-k]$。

因此，再用一次比较即可求得 $x[1:n]$中的次大数。

上述算法所需的比较次数 $T(n)$满足递归方程

$$T(n)=\begin{cases}1 & n=2\\ T\lceil n/2\rceil+\lfloor n/2\rfloor+1 & n>2\end{cases}$$

解此递归方程可得 $T(n)=n+\lceil \log n\rceil-2$。

由对手论证方法已证明该问题所需的比较次数至少为 $n+\lceil \log n\rceil-2$。因此，上述算法是求 n 个数中最大数与次大数的最优算法。

习题 2-17 整数集合排序

设 $S_1,S_2,\cdots,S_k$ 是整数集合，其中每个集合 S_i(其中 $1\leqslant i\leqslant k$) 中整数取值范围是 $1\sim n$，且 $\sum_{i=1}^{k}|S_i|=n$，试设计一个算法，在 $O(n)$时间内将 $S_1,S_2,\cdots,S_k$ 分别排序。

分析与解答：

用桶排序或基数排序算法思想可以实现整数集合排序。

习题 2-18 第 k 小元素问题的计算时间下界

试证明：在最坏情况下，求 n 个元素组成的集合 S 中的第 k 小元素至少需要 $n+\min(k,n-k+1)-2$ 次比较。

分析与解答：

由于要建立下界，不失一般性，可设 S 中的 n 个元素互不相同。首先注意到，要确定第 k 小元素 z，就要确定 S 中其他元素与 z 的关系。对于 S 中每个元素 x，必须确定 $x>z$ 或 $x<z$。换句话说，要建立 S 中元素与 z 的关系树，如图 2-5 所示。

图 2-5 中每个顶点表示一个元素，每一条边表示一次比较。较高的元素的值也较大。如果有一个元素 y 与第 k 小元素 z 的大小关系不确定，则对手可以改变 y 的值，使其从 z 的一侧移向另一侧，而不改变已做过比较的序的关系，从而改变了 z 的第 k 小的地位，产生矛盾，如图 2-6 所示。

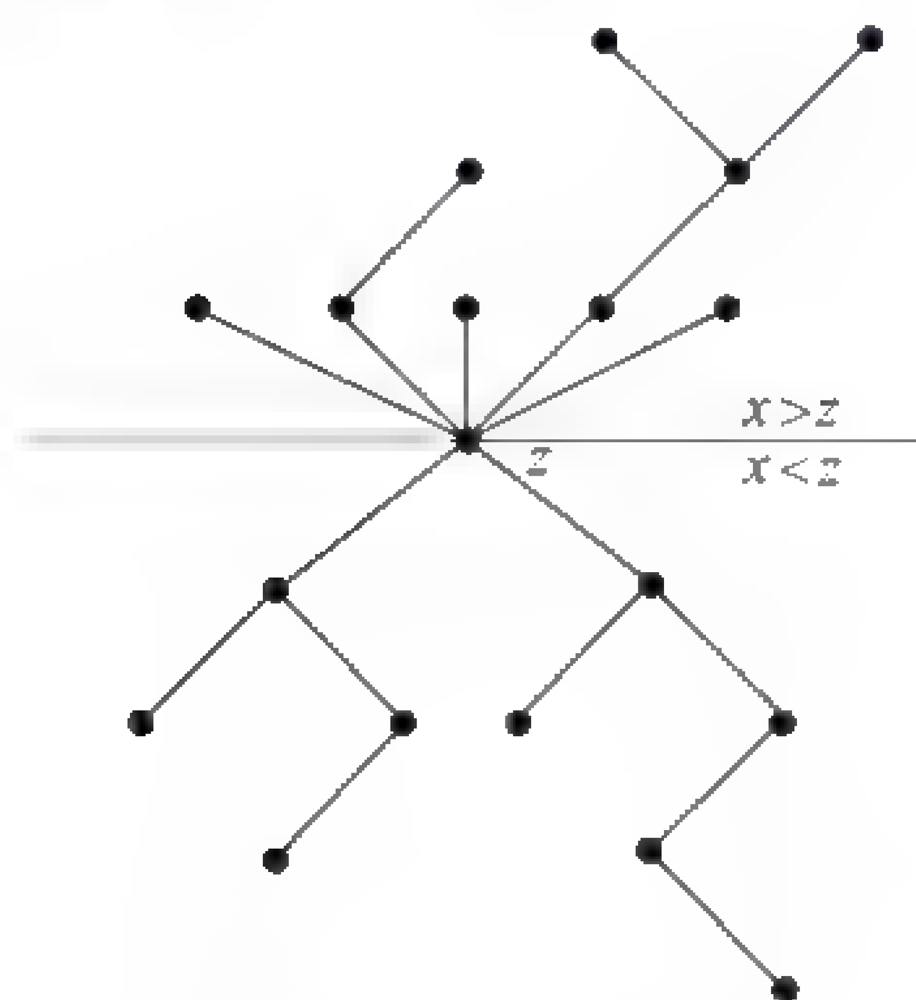


图 2-5 序关系树

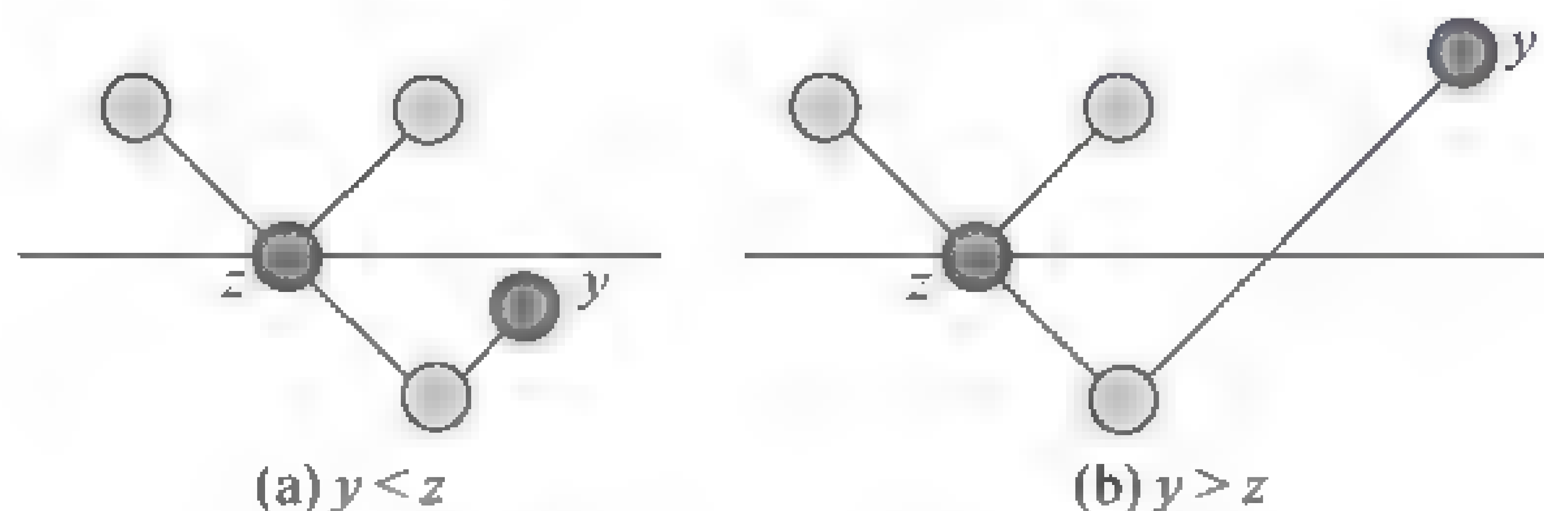


图 2-6 对手策略

由于关系树中有 n 个顶点，因此有 $n-1$ 条边，从而至少需要 $n-1$ 次比较。

下面进一步证明，采用对手论证方法，对手能迫使算法在得到所需的关系树中的 $n-1$ 次比较之前，进行其他“无用”的比较。

当 $y \geq z$ 时，两个元素 x 和 y 之间的第 1 次比较，$x > y$ 对应于关系树中的一条边。同理，当 $y \leq z$ 时，两个元素 x 和 y 之间的第 1 次比较，$x < y$ 也对应于关系树中的一条边。这类比较称为关键比较。反之，当 $x > z$ 且 $y < z$ 时，两个元素 x 和 y 之间的比较是非关键比较。

下面的对手策略将迫使算法进行尽可能多的非关键比较。对手首先选定第 k 小元素 z 的值，集合中其他元素的值未定，仅在算法进行与该元素有关的比较时确定该元素的值。在算法执行的任何阶段，集合中元素有下面 3 种状态。

L：该元素的值已确定，且大于 z。

S：该元素的值已确定，且小于 z。

N：该元素的值未确定。

算法对两个元素 x 和 y 进行比较时，对手根据元素 x 和 y 的状态，按照表 2-1 所示的对手策略确定状态为 N 的元素的值。

表 2-1　对手策略

x	y	对手策略
N	N	x 取值大于 z，y 取值小于 z
N	L	x 取值小于 z
L	N	y 取值小于 z
N	S	x 取值大于 z
S	N	y 取值大于 z

对手策略中的每个比较均为非关键比较。每个这类比较最多产生一个元素状态 L，且最多产生一个元素状态 S。算法最终产生 $k-1$ 个状态为 S 的元素和 $n-k$ 个状态为 L 的元素。因此，上述对手策略至少迫使算法做了 $\min\{k-1, n-k\}$ 次非关键比较。因此，任何算法至少做了 $n-1-\min\{k-1, n-k-1\}$ 次比较。由此可见，在最坏情况下，第 k 小元素问题至少需要 $n+\min(k, n-k+1)-2$ 次比较。

当 n 是奇数时，集合 S 的中位数是第 $(n+1)/2$ 小元素。由上述结论可知，$3n/2-3/2$ 是中位数问题的一个计算时间下界。

习题 2-19　非增序快速排序算法

如何修改主教材中算法 qSort 才能使其将输入元素按非增序排序？

分析与解答：

将算法 qSort 中的 partition 中的不等号反向即可。

```
private static int partition (int p, int r)
{
    int i=p, j=r + 1;
    Comparable x=a[p];
    while (true){
        // 将> x 的元素交换到左边区域
         while (a[++i].compareTo(x)>0);
        // 将< x 的元素交换到右边区域
         while (a[--j].compareTo(x) < 0);
         if (i >= j) break;
         MyMath.swap(a, i, j);
    }
    a[p]=a[j]; a[j]=x;
    return j;
}
```

习题 2-20　随机化算法

对一个随机化算法,为什么只分析其平均情况下的性能,而不分析其最坏情况下的性能?

分析与解答:

在随机化算法中,出现最坏情况下的实例往往是小概率事件。

习题 2-21　随机化快速排序算法

在执行 randomizedQuicksort 时,在最坏情况下,调用 random 多少次?在最好情况下又怎样?

分析与解答:

在最坏情况下,需要 $\Theta(n^2)$ 计算时间;在最好情况下,需要 $\Theta(n\log n)$ 计算时间。

习题 2-22　随机排列算法

试设计一个 $O(n)$ 时间算法,使之能产生数组 $a[0:n-1]$ 元素的一个随机排列。

分析与解答:

见主教材第 200 页算法 shuffle。

习题 2-23　算法 qSort 中的尾递归

试用 while 循环消去算法 qSort 中的尾递归,并比较消去尾递归前后算法的效率。

分析与解答:

消除尾递归的快速排序算法如下:

```
private static void qSort(int p, int r)
{
    while (p<r)
    {
        int q=partition(p,r);
        qSort(p,q-1);            // 对左半段排序
        p=q+1;                   // 对右半段排序
    }
}
```

习题 2-24　用栈模拟递归

试用栈来模拟递归,消去算法 qSort 中的递归,并证明所需的栈空间为 $O(\log n)$。

分析与解答:

对快速排序算法的另一个改进是模拟递归。当待排序数组 $a[l:r]$ 中有 n 个元素时,快速排序算法 qSort 的递归调用在最坏情况下可能耗费 $O(n)$ 栈空间。如果让左半段数组 $a[l:i-1]$ 和右半段数组 $a[i+1:r]$ 中元素个数较少者先排序,则在最坏情况下只耗费 $\log n$ 栈空间。事实上,设待排序数组大小为 n 时快速排序算法所需栈空间为 $s(n)$,若采用小者优先递归的策略,则 $s(n)$ 满足

$$s(n) \leqslant \begin{cases} 0 & n \leqslant 1 \\ s(n/2)+1 & n > 1 \end{cases}$$

由此可见,$s(n) \leqslant \log n$。

进一步采用模拟递归技术可以消去算法的递归调用。

```
static private void qSort(int l, int r)
```

```
{
    LinkedStack s=new LinkedStack();
    s.push (new Integer(r));s.push (new Integer(l));
    while (!s.empty())
        {
            l=((Integer)s.pop()).intValue();
            r =((Integer)s.pop()).intValue();
            if (r <= l) continue;
            int i=partition(l,r);
            if (i-l>r-i){
                s.push (new Integer(i-1));s.push (new Integer(l));
                s.push (new Integer(r));s.push (new Integer(i+1));
        }
        else
        {
            s.push (new Integer(r));s.push (new Integer(i+1));
            s.push (new Integer(i-1));s.push (new Integer(l));
        }
    }
}
```

习题 2-25　算法 select 中的元素划分

在算法 select 中，输入元素被划分为 5 个一组，如果将它们划分为 7 个一组，该算法仍然是线性时间算法吗？划分成 3 个一组又怎样？

分析与解答：

(1) 在算法 select 中，如果将输入元素划分为 7 个一组，则大于或等于 x 的元素个数至少为 $4\left(\left\lceil\frac{1}{2}\lceil n/7\rceil\right\rceil-2\right)\geq\frac{2n}{7}-8$。因此，所需计算时间 $T(n)$ 满足递归式

$$T(n)\leqslant T(\lceil n/7\rceil)+T(5n/7+8)+O(n)$$

解此递归式可得 $T(n)=O(n)$，即按 7 个一组划分算法仍然是线性时间算法。

(2) 如果将输入元素划分为 3 个一组，则 $T(n)$ 所满足的递归式为

$$T(n)\leqslant T(\lceil n/3\rceil)+T(2n/3+4)+O(n)$$

解此递归式可得 $T(n)=O(n\log n)$。

事实上，在这种分组原则下，可以构造出最坏情形输入，使 $T(n)=\Theta(n\log n)$。这说明如果将输入划分为 3 个一组，算法就不再是线性时间算法了。

(3) 从上面的讨论容易得到一般的结论。如果将输入划分为 k 个一组，其中 k 是不小于 5 的奇数时，算法 select 仍然为线性时间算法。

习题 2-26　$O(n\log n)$时间快速排序算法

试说明如何修改快速排序算法，使它在最坏情况下的计算时间为 $O(n\log n)$。

分析与解答：

快速排序算法的效率在很大程度上取决于划分基准 x 的选取。如果每次选取的划分基准能将待划分子数组分为大小基本相同的两个子数组，就满足了分治法的平衡原则，从而可以保证在最坏情况下，算法 qSort 的计算时间为 $O(n\log n)$。由此，容易想到如果能在线性时间内取得子数组 $A[p:r]$的中位数 median，以 median 作为划分基准就可以达到目的。这

里的线性时间选择算法 select 起关键作用。

习题 2-27　最接近中位数的 k 个数

给定由 n 个互不相同的数组成的集合 S 以及正整数 $k \leqslant n$，试设计一个 $O(n)$ 时间算法，找出 S 中最接近 S 的中位数的 k 个数。

分析与解答：

(1) 找出 S 的中位数 median。

(2) 计算 $T=\{|x-\text{median}| \mid x\in S\}$。

(3) 找出 T 的第 k 小元素 y。

(4) 根据 y 找出所要的解 $\{x\in S \mid |x-\text{median}|\leqslant y\}$。

由于在最坏情况下 select 的计算时间为 $O(n)$，因此(1)和(3)需要 $O(n)$ 时间。(2)和(4)显然只需要 $O(n)$时间。因此，在最坏情况下算法所需的计算时间为 $O(n)$。

习题 2-28　X 和 Y 的中位数

设 $X[0:n-1]$和 $Y[0:n-1]$为两个数组，每个数组中含有 n 个已排好序的数。试设计一个 $O(\log n)$时间的算法，找出 X 和 Y 的 $2n$ 个数的中位数。

分析与解答：

1) 算法设计思想

考虑稍一般的问题：设 $X[i_1:j_1]$和 $Y[i_2:j_2]$是 X 和 Y 的排好序的子数组，且长度相同，即 $j_1-i_1=j_2-i_2$。找出这两个子数组中 $2(j_1-i_1+1)$个数的中位数。

首先注意到，若 $X[i_1]\leqslant Y[j_2]$，则中位数 median 满足 $X[i_1]\leqslant \text{median}\leqslant Y[j_2]$。同理，若 $X[i_1]\geqslant Y[j_2]$，则 $Y[j_2]\leqslant \text{median}\leqslant X[i_1]$。

设 $m_1=(i_1+j_1)/2, m_2=(i_2+j_2)/2$，则 $m_1+m_2=(i_1+j_1)/2+(i_2+j_2)/2=i_1+(j_1-i_1)/2+i_2+(j_2-i_2)/2=i_1+i_2+(j_1-i_1)/2+(j_2-i_2)/2$。

由于 $j_1-i_1=j_2-i_2$，故$(j_1-i_1)/2+(j_2-i_2)/2=j_1-i_1=j_2-i_2$。

因此，$m_1+m_2=i_1+i_2+j_1=i_1=i_2+j_1=i_1+i_2+j_2-i_2=i_1+j_2$。

当 $X[m_1]=Y[m_2]$时，$\text{median}=X[m_1]=Y[m_2]$。

当 $X[m_1]<Y[m_2]$时，设 median_1 是 $X[m_1:j_1]$和 $Y[j_2:m_2]$的中位数，则 $\text{median}=\text{median}_1$。

当 $X[m_1]>Y[m_2]$时，设 median_2 是 $X[i_1:m_1]$和 $Y[m_2:j_2]$的中位数，类似地有 $\text{median}=\text{median}_2$。

通过以上的讨论，可以设计出找两个子数组 $X[i_1:j_1]$和 $Y[i_2:j_2]$的中位数 median 的算法。

2) 算法复杂性

设在最坏情况下，算法所需的计算时间为 $T(2n)$。由算法中 m_1 和 m_2 的选取策略可知，在递归调用时，数组 X 和 Y 的大小都减少了一半。因此，$T(2n)$ 满足递归式

$$T(2n)=\begin{cases}O(1) & n<c\\ T(n)+O(1) & n\geqslant c\end{cases}$$

解此递归方程可得 $T(2n)=O(\log n)$。

习题 2-29　网络开关设计

考查如图 2-7 所示的有两个输入端和两个输出端的位置 2 开关。当开关处于位置 1 时，输入 1 和 2 分别产生输出 1 和 2；当开关处于位置 2 时，输入 1 和 2 分别产

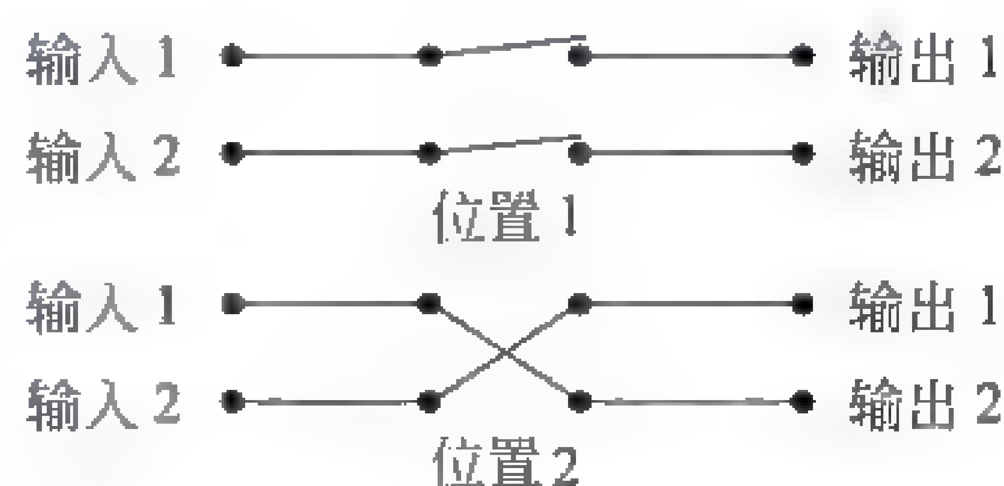


图 2-7　2 位置开关

生输出2和1。使用这种开关设计一个有 n 个输入端和 n 个输出端的开关网络，实现将输入的 n 个数值以它们的 $n!$ 种不同排列的任何一种排列输出(通过开关位置的适当选择)。要求网络中使用的开关个数为 $O(n\log n)$。

分析与解答：

为简明起见，不妨设 $n=2^k$。用分治法递归地构造所需的开关网络 P_n。在开关网络 P_n 的输入端和输出端分别使用 $n/2$ 个开关进行连接。这些开关的输入端和输出端分别与两个能产生 $1,2,\cdots,n/2$ 的任意排列的开关网络 $P_{n/2}$ 相连。在这个开关网络 P_n 中，输入端和输出端的每一个开关都有一条线与上面的开关网络 $P_{n/2}$ 连接，且另一条线与下面的开关网络 $P_{n/2}$ 相连。

(1) 对于 $1,2,\cdots,n$ 的任意一个排列 π，可以通过对上述开关网络 P_n 的输入端和输出端的 n 个开关进行设置，使得输入端的 i 与输出端的 $\pi(i)$ 连接到同一个 $P_{n/2}$ 开关网络。

事实上，将输出端和输入端的 n 个端口从上到下分别记为 u_i 和 v_i，$1\leqslant i\leqslant n$，并将这些端口看作图 G 中的顶点，构造图 $G=(V,E)$ 如下：

设 $V_1=\{u_i,1\leqslant i\leqslant n\}$，$V_2=\{v_i,1\leqslant i\leqslant n\}$，$V=V_1\cup V_2$。

设 $EU=\{(u_{2i-1},u_{2i}),1\leqslant i\leqslant n/2\}$，$EV=\{(v_{2i-1},v_{2i}),1\leqslant i\leqslant n/2\}$，

$EUV=\{(u_i,v_{\pi(i)}),1\leqslant i\leqslant n\}$，取 $E=EU\cup EV\cup EUV$。

由于 i 与 $\pi(i)$ 一一对应，故图 G 中每个顶点的度均为2。因此，图 G 的每一个连接分支都是一个欧拉回路。对图 G 的任一连通分支 G_i，设它的一条欧拉回路 t 为 $t_1,t_2,\cdots,t_{2k-1},t_{2k},t_1$，且 $t_1=u_i$。对各顶点 t_i 作0-1赋值 $f(t_i)$ 如下：

$$f(t_i)=\begin{cases}0 & i=1\\ f(t_{i-1}) & (t_{i-1},t_i)\in EUV\\ \overline{f(t_{i-1})} & \text{否则}\end{cases}$$

容易看出这种0-1赋值是唯一确定的。根据这个0-1赋值，将 $f(t_i)=0$ 的顶点 t_i 所对应的端口连接到上面的开关网络 $P_{n/2}$；将 $f(t_i)=1$ 的顶点 t_i 所对应的端口连接到下面的开关网络 $P_{n/2}$。这个连接原则对应于输入端与输出端的 n 个开关的一种确定的设置，即当 $f(u_{2i-1})=0$ 时，对应于输出端第 i 个开关的平行连接设置，$1\leqslant i\leqslant n/2$；当 $f(u_{2i-1})=1$ 时，对应于输出端第 i 个开关的交叉连接设置。输入端的开关设置是类似的。当 $(u_i,v_{\pi(i)})\in t$ 时，$f(u_i)=f(v_{\pi(i)})$，这说明 u_i 与 $v_{\pi(i)}$ 连接到同一个 $P_{n/2}$ 开关网络。

(2) 由 $f(t_i)$ 的值确定了输入端与输出端的 n 个开关的设置后，将排列的任务交给两个 $P_{n/2}$ 开关网络去完成。递归地构造出 $P_{n/2},P_{n/4},\cdots$，直至构造出整个开关网络 P_n。

(3) 设 P_n 所需的开关个数为 $T(n)$，由上述分治递归构造过程可知，$T(n)$ 满足如下递归方程

$$T(n)=\begin{cases}1 & n=2\\ 2T(n/2)+n & n>2\end{cases}$$

解此递归方程可得 $T(n)=n\log n-n/2$。

习题 2-30　带权中位数问题

对于 n 个带有正权 $w_1,w_2,\cdots,w_n$，且 $\sum_{i=1}^{n}w_i=1$ 的互不相同的元素 $x_1,x_2,\cdots,x_n$，其带权中位数 x_k 满足

$$\begin{cases} \sum\limits_{x_i < x_k} w_i \leqslant \dfrac{1}{2} \\ \sum\limits_{x_i > x_k} w_i \leqslant \dfrac{1}{2} \end{cases}$$

(1) 试证明 $x_1, x_2, \cdots, x_n$ 的不带权中位数是带权 $w_i = 1/n$, $i = 1,2,\cdots,n$ 的带权中位数。

(2) 说明如何通过排序,在最坏情况下,用 $O(n\log n)$ 时间求出 n 个元素的带权中位数。

(3) 说明如何利用一个线性时间选择算法(如 select),在最坏情况下,用 $O(n)$ 时间求出 n 个元素的带权中位数。

(4) 邮局位置问题定义为:已知 n 个点 $p_1, p_2, \cdots, p_n$ 以及与它们相联系的权 $w_1, w_2, \cdots, w_n$,要求确定一点 p(p 不一定是 n 个输入点之一),使和式 $\sum\limits_{i=1}^{n} w_i d(p, p_i)$ 达到最小,其中 $d(a,b)$ 表示 a 与 b 之间的距离。

试论证带权中位数是一维邮局问题的最优解。此时,$d(a,b)=|a-b|$。

(5) 在二维的情形如何找邮局问题的最优解?

分析与解答:

(1) 取 $w_i = \dfrac{1}{n}, 1 \leqslant i \leqslant n$,则

$$\begin{cases} \sum\limits_{x_i < x_k} \dfrac{1}{n} \leqslant \dfrac{1}{2} \\ \sum\limits_{x_i > x_k} \dfrac{1}{n} \leqslant \dfrac{1}{2} \end{cases}$$

当且仅当

$$\begin{cases} \sum\limits_{x_i < x_k} 1 \leqslant \dfrac{n}{2} \\ \sum\limits_{x_i > x_k} 1 \leqslant \dfrac{n}{2} \end{cases}$$

当且仅当 x_k 是 $x_i, 1 \leqslant i \leqslant n$ 的中位数。

(2) 将 $x_i, 1 \leqslant i \leqslant n$ 从小到大排序需要 $O(n\log n)$ 时间。对排好序的序列做一次线性扫描即可找出带权中位数。

(3) 用线性时间选择算法 select 和分治策略,可设计 $O(n)$ 时间求带权中位数算法。

(4) 设 x^* 是一维邮局问题的最优解,可以证明 $x^* = x_k$。x_k 是 x_i(其中 $1 \leqslant i \leqslant n$)的带权 w_i(其中 $1 \leqslant i \leqslant n$)的带权中位数。

(5) 在二维的情形,平面上两个点 $p=(x_p, y_p)$ 和 $q=(x_q, y_q)$ 之间的距离定义为:

$$d(p,q) = |x_p - x_q| + |y_p - y_q|$$

若平面上 n 个点 p_i(其中 $1 \leqslant i \leqslant n$)分别带权 w_i(其中 $1 \leqslant i \leqslant n$),则二维邮局问题求平面上一确定点 p,使和式 $\sum\limits_{i=1}^{n} w_i d(p, p_i)$ 达到最小。

此问题显然可以转化为两个一维邮局问题,即分别求 x_i(其中 $1 \leqslant i \leqslant n$)的带权 w_i(其中 $1 \leqslant i \leqslant n$)的一维邮局问题的解 x 和 y_i(其中 $1 \leqslant i \leqslant n$)的带权 w_i(其中 $1 \leqslant i \leqslant n$)的一维邮局问题的解 y,则 $p=(x, y)$ 即为二维邮局问题的解。由上面的讨论及(3)和(4)即知

可在$O(n)$时间内解二维邮局问题。

习题 2-31　构造 Gray 码的分治算法

Gray 码是一个长度为2^n的序列。序列中无相同元素，每个元素都是长度为n位的(0,1)串，相邻元素恰好只有1位不同。用分治策略设计一个算法，对任意的n构造相应的Gray码。

分析与解答：

考查$n=1,2,3$的简单情形。

$n=1$			
0	1		
$n=2$			
00	01		
11	10		
$n=3$			
000	001	011	010
110	111	101	100

设n位 Gray 码序列为$G(n)$，将$G(n)$以相反顺序排列的序列为$G^{-1}(n)$。从上面的简单情形可以看出$G(n)$的构造规律：$G(n+1)=0G(n)1G^{-1}(n)$。

注意到$G(n)$的最后一个n位串与$G^{-1}(n)$的第1个n位串相同，可用数学归纳法证明$G(n)$的上述构造规律。由此规律容易设计构造$G(n)$的分治法如下：

```
public static void Gray(int n)
{
    if (n==1){a[1]=0;a[2]=1;return;}
    Gray(n-1);
    for(int k=1<<(n-1),i=k;i>0;i--)a[2*k-i+1]=a[i]+k;
}
```

上述算法中将n位(0,1)串看作是二进制整数，第i个串存储在$a[i]$中。

完成计算后，由 out 输出 Gray 码序列。

```
public static void out(int n)
{
    int m=1<<n;
    for (int i=1; i<=m; i++)
    {
        String str=Integer.toBinaryString (a[i]);
        int s=str.length();
        for(int j=0;j<n-s;j++)System.out.print("0");
        System.out.println(Integer.toBinaryString (a[i])+" ");
    }
    System.out.println();
```

```
}
```

习题 2-32　网球循环赛日程表

设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表：

(1) 每个选手必须与其他 $n-1$ 个选手各赛一次。

(2) 每个选手一天只能赛一次。

(3) 当 n 是偶数时循环赛进行 $n-1$ 天，当 n 是奇数时循环赛进行 n 天。

分析与解答：

方法 1：分治法

主教材中的分治法应描述如下：

```
public static void tourna(int n)
{
    if (n==1) {a[1][1]=1;return;}
    tourna(n/2);
    copy(n);
}
```

算法 copy 将左上角递归计算出的小块中的所有数字，按其相对位置抄到右下角；将左上角小块中的所有数字加 $n/2$ 后，按其相对位置抄到左下角；将左下角小块中的所有数字，按其相对位置抄到右上角，这样就完成了比赛日程表。

```
public static void copy(int n)
{
    int m=n/2;
    for(int i=1;i<=m;i++)
        for(int j=1;j<=m;j++)
        {
            a[i][j+m]=a[i][j]+m;
            a[i+m][j]=a[i][j+m];
            a[i+m][j+m]=a[i][j];
        }
}
```

对于一般的正整数 n，当 n 是奇数时，增设一个虚拟选手 $n+1$，将问题转换为 n 是偶数的情形。当选手遇到与虚拟选手比赛时，表示轮空。因此，只要关注 n 为偶数的情形。

当 $n/2$ 为偶数时，与 $n=2k$ 的情形类似，可用分治法求解。

当 $n/2$ 为奇数时，递归返回的轮空的比赛要做进一步处理。其中一种处理方法是在前 $n/2$ 轮比赛中轮空选手与下一个未参赛选手进行比赛。

一般情况下的分治法 tournament 可描述如下：

```
public static void tournament(int n)
{
    if (n==1) {a[1][1]=1;return;}
    if (odd(n)){tournament(n+1);return;}
    tournament(n/2);
```

```
    makecopy(n);
}
public static boolean odd(int n)
{
    return (n%2>0);
}
```

算法 makecopy 与算法 copy 类似，但要区分 $n/2$ 为奇数或偶数的情形。

```
public static void makecopy(int n)
{
    if (n/2>1 && odd(n/2)) copyodd(n);
    else copy(n);
}
```

算法 copyodd 实现 $n/2$ 为奇数时的复制。

```
public static void copyodd(int n)
{
    int m=n/2;
    for(int i=1;i<=m;i++)
    {
        b[i]=m+i;b[m+i]=b[i];
    }
    for(int i=1;i<=m;i++)
    {
        for(int j=1;j<=m+1;j++)
        {
            if (a[i][j]>m) {a[i][j]=b[i];a[m+i][j]=(b[i]+m)%n;}
            else a[m+i][j]=a[i][j]+m;
        }
        for(int j=2;j<=m;j++)
        {
            a[i][m+j]=b[i+j-1];
            a[b[i+j-1]][m+j]=i;
        }
    }
}
```

用上述算法计算出的 $n=10$ 的比赛日程表如表 2-2 所示。

表 2-2　用分治法计算出的 $n=10$ 的比赛日程表

1	2	3	4	5	6	7	8	9	10
2	1	5	3	7	4	8	9	10	6
3	8	1	2	4	5	9	10	6	7
4	5	9	1	3	2	10	6	7	8

续表

5	4	2	10	1	3	6	7	8	9
6	7	8	9	10	1	5	4	3	2
7	6	10	8	2	9	1	5	4	3
8	3	6	7	9	10	2	1	5	4
9	10	4	6	8	7	3	2	1	5
10	9	7	5	6	8	4	3	2	1

方法 2：多边形方法

n 是偶数的情形。

循环赛进行 $n-1$ 天，每个选手与其他 $n-1$ 个选手各赛一次。

用一个 $n-1$ 边的正多边形表示 1 轮比赛。多边形的顶点和中心点表示参赛选手。当 $n=8$ 时的循环赛多边形如图 2-8 所示。

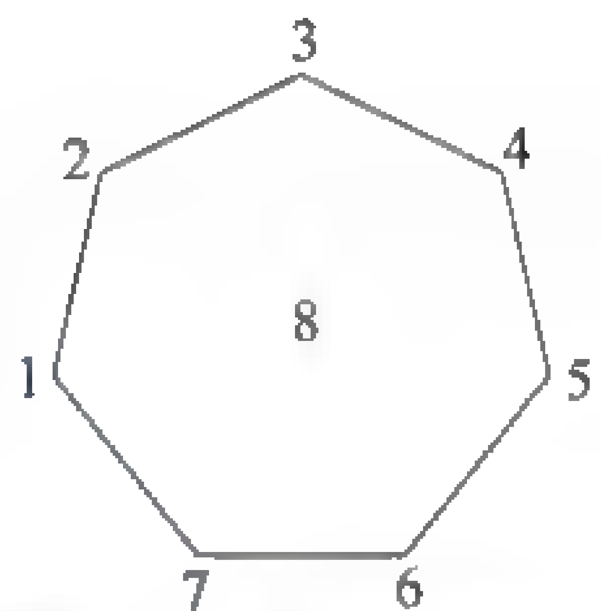


图 2-8　循环赛多边形

用水平线连接循环赛多边形的 $n-2$ 个顶点，并将剩下的那个顶点与中心点连接，如图 2-9 所示。每一条连线表示一场比赛。

图 2-9 所表示的第 1 轮比赛的场次是：(7,6)，(1,5)，(2,4)和 (3,8)。

将多边形绕中心顺时针旋转 $2\pi/(n-1)$ 弧度得到新的循环赛多边形，如图 2-10 所示。

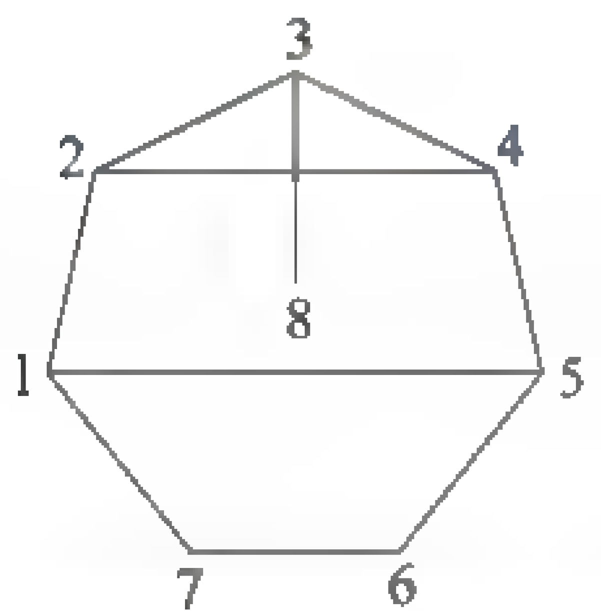


图 2-9　顶点间的连线表示比赛场次

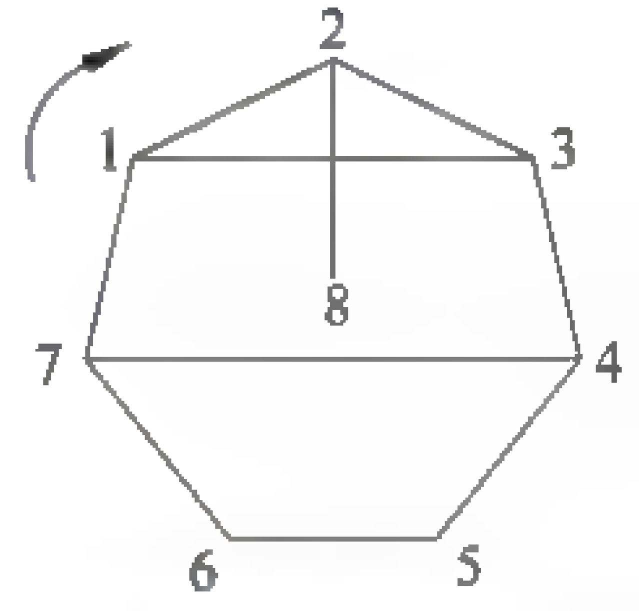


图 2-10　顺时针旋转 $2\pi/(n-1)$ 弧度

由此得到新一轮比赛的场次是：(6,5)，(7,4)，(1,3)和(2,8)。

按此方式可旋转多边形 $n-2$ 次。后继的旋转如图 2-11 所示。

n 是奇数的情形同样可转换为 n 是偶数的情形。

按照上述思想实现的构造法如下：

```
public static void construct(int n)
{
    if (n==1) return;
    int m=odd(n)? n:n-1;
    a[n][1]=n;
    for(int i=1;i<=m;i++)
    {
```

```
        a[i][1]=i;b[i]=i+1;b[m+i]=i+1;
    }
    for(int i=1;i<=m;i++)
    {
        a[1][i+1]=b[i];a[b[i]][i+1]=1;
        for(int j=1;j<=m/2;j++)
        {
            int k=b[i+j], r=b[i+m-j];
            a[k][i+1]=r;a[r][i+1]=k;
        }
    }
}
```

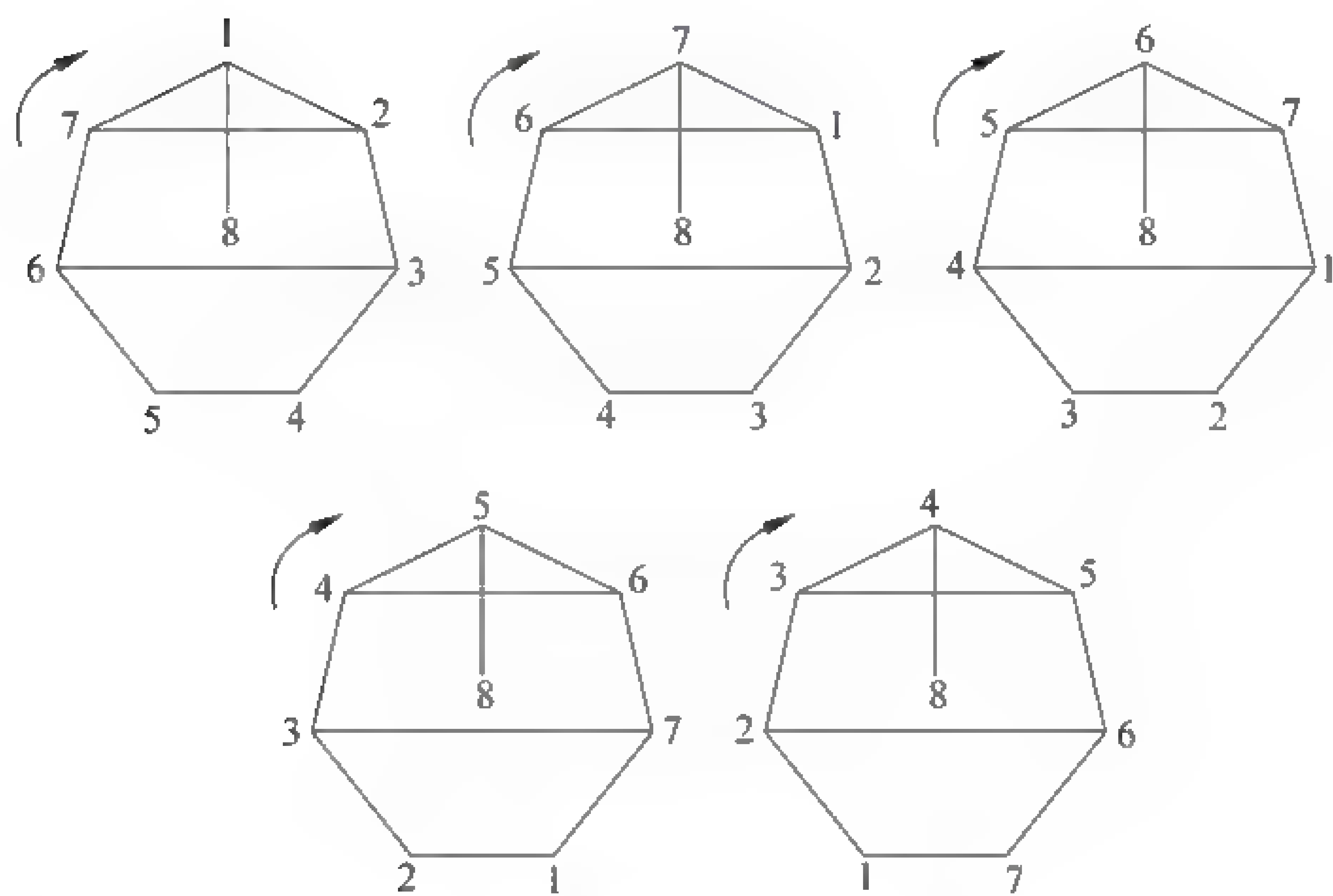


图 2-11　顺时针旋转多边形 $n-2$ 次

用上述算法计算出的 $n=10$ 的比赛日程如表 2-3 所示。

表 2-3　用多边形方法计算出的 $n=10$ 的比赛日程表

1	2	3	4	5	6	7	8	9	10
2	1	4	6	8	10	3	5	7	9
3	10	1	5	7	9	2	4	6	8
4	9	2	1	6	8	10	3	5	7
5	8	10	3	1	7	9	2	4	6
6	7	9	2	4	1	8	10	3	5
7	6	8	10	3	5	1	9	2	4
8	5	7	9	2	4	6	1	10	3
9	4	6	8	10	3	5	7	1	2
10	3	5	7	9	2	4	6	8	1

用上述算法计算出的 $n=8$ 的比赛日程表完全图如图 2-12 所示。

算法实现题 2-1　输油管道问题

★ 问题描述

某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 口油井的油田。从每口油井都要有一条输油管道沿最短路径(或南或北)与主管道相连。如果给定 n 口油井的位置,即它们的 x 坐标(东西向)和 y 坐标(南北向),应如何确定主管道的最优位置(即使各油井到主管道之间的输油管道长度总和最小的位置)?证明可在线性时间内确定主管道的最优位置。

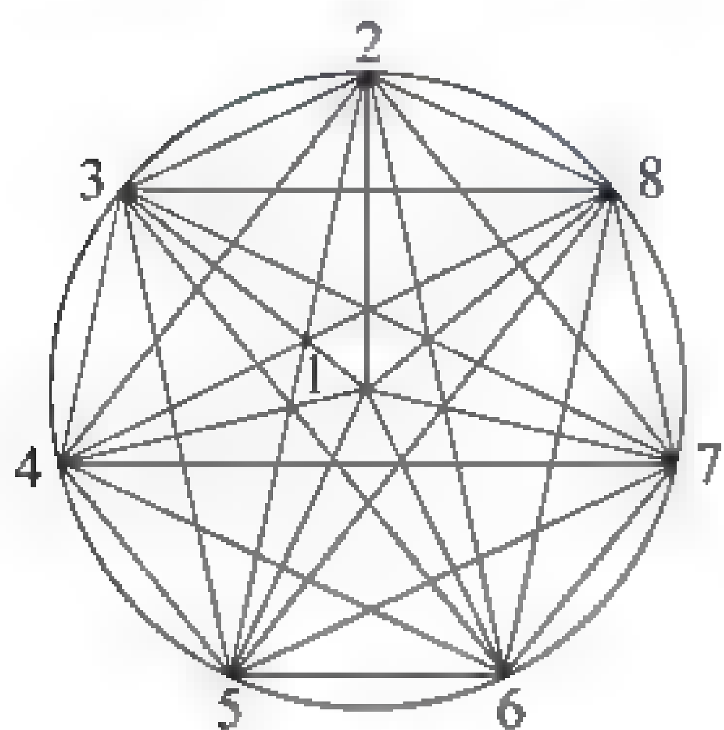


图2-12　循环赛日程表 n=8 的解

★ 算法设计

给定 n 口油井的位置,计算各油井到主管道之间的输油管道最小长度总和。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是油井数 n,$1 \leqslant n \leqslant 10\,000$。接下来 n 行是油井的位置,每行两个整数 x 和 y,$-10\,000 \leqslant x, y \leqslant 10\,000$。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第1行中的数是油井到主管道之间的输油管道最小长度总和。

输入文件示例	输出文件示例
input.txt	output.txt
5	6
1 2	
2 2	
1 3	
3 −2	
3 3	
6	

分析与解答:

设 n 口油井的位置分别为 $p_i=(x_i, y_i)$,$1 \leqslant i \leqslant n$。由于主输油管道是东西向的,因此可用其主轴线的 y 坐标唯一确定其位置。主管道的最优位置 y 应使 $\sum_{i=1}^{n} d(y, y_i)$ 达到最小,其中,$d(y, y_i)=|y-y_i|$。这正是习题 2-30 中一维邮局问题的特殊情形,即 $w_i=\frac{1}{n}$(其中 $1 \leqslant i \leqslant n$)的情形。由带权中位数问题的解答可知,$y_i$(其中 $1 \leqslant i \leqslant n$)的中位数 y_k 即为输油管道问题的最优解。用任一线性时间找中位数算法,均可在 $O(n)$ 时间内解此问题。

算法实现题 2-2　众数问题

★ 问题描述

给定含有 n 个元素的多重集合 S,每个元素在 S 中出现的次数称为该元素的重数。多重集合 S 中重数最大的元素称为众数。

例如，$S=\{1,2,2,2,3,5\}$。

多重集合 S 的众数是2，其重数为3。

★ **算法设计**

对于给定的由 n 个自然数组成的多重集合 S，计算 S 的众数及其重数。

★ **数据输入**

输入数据由文件名为 input.txt 的文本文件提供。

文件的第1行为多重集合 S 中元素个数 n；接下来的 n 行中，每行有1个自然数。

★ **结果输出**

将计算结果输出到文件 output.txt 中。输出文件有两行，第1行给出众数，第2行是重数。

输入文件示例	输出文件示例
input.txt	output.txt
6	2
1	3
2	
2	
2	
3	
5	
2	
3	

分析与解答：

解众数问题的分治算法实现如下：

```
public static void mode(int l, int r)
{
    int []lr=new int[2];
    int med=median(a,l,r);
    split(a,med,l,r,lr);
    if(largest<lr[1]-lr[0]+1){ largest=lr[1]-lr[0]+1;element=med;}
    if (lr[0]-l>largest) mode(l,lr[0]-1);
    if (r-lr[1]>largest) mode(lr[1]+1,r);
```

其中，median 用于找中位数。split 用中位数将数组分割为2段。

算法实现题 2-3　邮局选址问题

★ **问题描述**

在一个按照东西和南北方向划分成规整街区的城市里，n 个居民点散乱地分布在不同的街区中。用 x 坐标表示东西向，用 y 坐标表示南北向。各居民点的位置可以由坐标 (x,y) 表示。街区中任意两点 (x_1,y_1) 和 (x_2,y_2) 之间的距离可以用数值 $|x_1-x_2|+|y_1-y_2|$ 度量。

居民们希望在城市中选择建立邮局的最佳位置，使 n 个居民点到邮局的距离总和最小。

★ **算法设计**

给定 n 个居民点的位置，计算 n 个居民点到邮局的距离总和的最小值。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是居民点数 n，$1\leqslant n\leqslant 10\,000$。接下来 n 行是居民点的位置，每行两个整数 x 和 y，$-10\,000\leqslant x,y\leqslant 10\,000$。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第1行中的数是 n 个居民点到邮局的距离总和的最小值。

输入文件示例	输出文件示例
input.txt	output.txt
5	10
1 2	
2 2	
1 3	
3 −2	
3 3	

分析与解答：

见习题2-30带权中位数问题。

算法实现题 2-4　马的 Hamilton 周游路线问题

★ 问题描述

8×8的国际象棋棋盘上的一匹马，恰好走过除起点外的其他63个位置各一次，最后回到起点。这条路线称为一条马的 Hamilton 周游路线。对于给定的 $m\times n$ 的国际象棋棋盘，m 和 n 均为大于5的偶数，且 $|m-n|\leqslant 2$，试设计一个分治算法找出一条马的 Hamilton 周游路线。

★ 算法设计

对于给定的偶数 $m,n\geqslant 6$，且 $|m-n|\leqslant 2$，计算 $m\times n$ 的国际象棋棋盘一条马的 Hamilton 周游路线。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有两个正整数 m 和 n，表示给定的国际象棋棋盘由 m 行、每行 n 个格子组成。

★ 结果输出

将计算出的马的 Hamilton 周游路线，用下面的两种表达方式输出到文件 output.txt 中。

第1种表达方式，按照马步的次序给出马的 Hamilton 周游路线。马的每一步用所在的方格坐标 (x,y) 来表示。x 表示行的坐标，编号为 $0,1,\cdots,m-1$；y 表示列的坐标，编号为 $0,1,\cdots,n-1$。起始方格为(0,0)。

第2种表达方式，在棋盘的方格中标明马到达该方格的步数。(0,0)方格为起跳步，并标明为第1步。

输入文件示例	输出文件示例
input.txt	output.txt
6 6	(0,0) (2,1) (4,0) (5,2) (4,4) (2,3)
	(0,4) (2,5) (1,3) (0,5) (2,4) (4,5)

(5,3) (3,2) (5,1) (3,0) (1,1) (0,3)
(1,5) (3,4) (5,5) (4,3) (3,1) (5,0)
(4,2) (5,4) (3,5) (1,4) (0,2) (1,0)
(2,2) (0,1) (2,0) (4,1) (3,3) (1,2)

1 32 29 18 7 10
30 17 36 9 28 19
33 2 31 6 11 8
16 23 14 35 20 27
3 34 25 22 5 12
24 15 4 13 26 21

分析与解答：

1) 算法思想

在 $n\times n$ 的国际象棋棋盘上的一匹马，可按 8 个不同方向移动。定义 $n\times n$ 的国际象棋棋盘上的马步图为 $G=(V,E)$。棋盘上的每个方格对应于图 G 中的一个顶点，$V=\{(i,j)|0\leqslant i,j<n\}$。从一个顶点到另一个马步可跳达的顶点之间有一条边。$E=\{((u,v),(s,t))|\{|u-s|,|v-t|\}=\{1,2\}\}$。

图 G 有 n^2 个顶点和 $4n^2-12n+8$ 条边。马的 Hamilton 周游路线问题即图 G 的Hamilton 回路问题。容易看出，当 n 为奇数时该问题无解。事实上，由于马在棋盘上移动的方格是黑白相间的，如果有解，则走到的黑白格子数相同，因此棋盘格子总数应为偶数，然而 n^2 为奇数，此为矛盾。下面给出的算法可以证明，当 $n\geqslant 6$ 是偶数时，问题有解，而且可以用分治法在线性时间内构造出一个解。

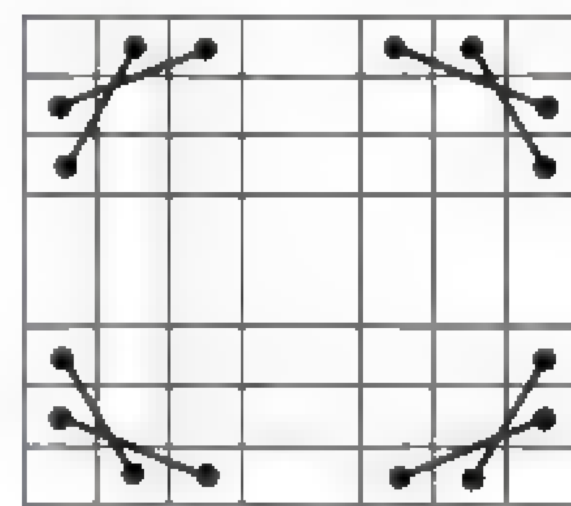


图 2-13 结构化的 Hamilton 回路

考查稍一般的情况，即给定的国际象棋棋盘有 m 行和 n 列，且 $|m-n|\leqslant 2$ 的情况。因此，可能有 $m\times m$，$m\times(m-2)$ 和 $m\times(m+2)$ 这 3 种不同规格的棋盘。为了采用分治策略，考查一类具有特殊结构的解，这类解在棋盘的 4 个角都包含 2 条特殊的边，如图 2-13 所示。称具有这类特殊结构的 Hamilton 回路为结构化的 Hamilton 回路。

用回溯法可在 $O(1)$ 时间内找出 6×6，6×8，8×8，8×10，10×10，10×12 棋盘上的结构化的 Hamilton 回路，如图 2-14 所示。

1	30	33	16	3	24
32	17	2	23	34	15
29	36	31	14	25	4
18	9	6	35	22	13
7	28	11	20	5	26
10	19	8	27	12	21

(a) 6×6 棋盘上的结构化 Hamilton 回路

1	10	31	40	21	14	29	38
32	41	2	11	30	39	22	13
9	48	33	20	15	12	37	28
42	3	44	47	6	25	18	23
45	8	5	34	19	16	27	36
4	43	46	7	26	35	24	17

(b) 6×8 棋盘上的结构化 Hamilton 回路

图 2-14 6×6，6×8，8×8，8×10，10×10，10×12 棋盘上的结构化 Hamilton 回路

1	46	17	50	3	6	31	52
18	49	2	7	30	51	56	5
45	64	47	16	27	4	53	32
48	19	8	29	10	55	26	57
63	44	11	22	15	28	33	54
12	41	20	9	36	23	58	25
43	62	39	14	21	60	37	34
40	13	42	61	38	35	24	59

(c) 8×8 棋盘上的结构化 Hamilton 回路

1	46	37	66	3	48	35	68	5	8
38	65	2	47	36	67	4	7	34	69
45	80	39	24	49	18	31	52	9	6
64	23	44	21	30	15	50	19	70	33
79	40	25	14	17	20	53	32	51	10
26	63	22	43	54	29	16	73	58	71
41	78	61	28	13	76	59	56	11	74
62	27	42	77	60	55	12	75	72	57

(d) 8×10 棋盘上的结构化 Hamilton 回路

1	54	69	66	3	56	39	64	5	8
68	71	2	55	38	65	4	7	88	63
53	100	67	70	57	26	35	40	9	6
72	75	52	27	42	37	58	87	62	89
99	30	73	44	25	34	41	36	59	10
74	51	76	31	28	43	86	81	90	61
77	98	29	24	45	80	33	60	11	92
50	23	48	79	32	85	82	91	14	17
97	78	21	84	95	46	19	16	93	12
22	49	96	47	20	83	94	13	18	15

(e) 10×10 棋盘上的结构化 Hamilton 回路

1	4	119	100	65	6	69	102	71	8	75	104
118	99	2	5	68	101	42	7	28	103	72	9
3	120	97	64	41	66	25	70	39	74	105	76
98	117	48	67	62	43	40	27	60	29	10	73
93	96	63	44	47	26	61	24	33	38	77	106
116	51	94	49	20	23	46	37	30	59	34	11
95	92	115	52	45	54	21	32	35	80	107	78
114	89	50	19	22	85	36	55	58	31	12	81
91	18	87	112	53	16	57	110	83	14	79	108
88	113	90	17	86	111	84	15	56	109	82	13

(f) 10×12 棋盘上的结构化 Hamilton 回路

图 2-14 (续)

图 2-14 中的 6×8,8×10 和 10×12 棋盘上的结构化 Hamilton 回路旋转 90°,可以得到 8×6,10×8 和 12×10 棋盘上的结构化 Hamilton 回路。

在棋盘上画出的结构化 Hamilton 回路如图 2-15 所示。

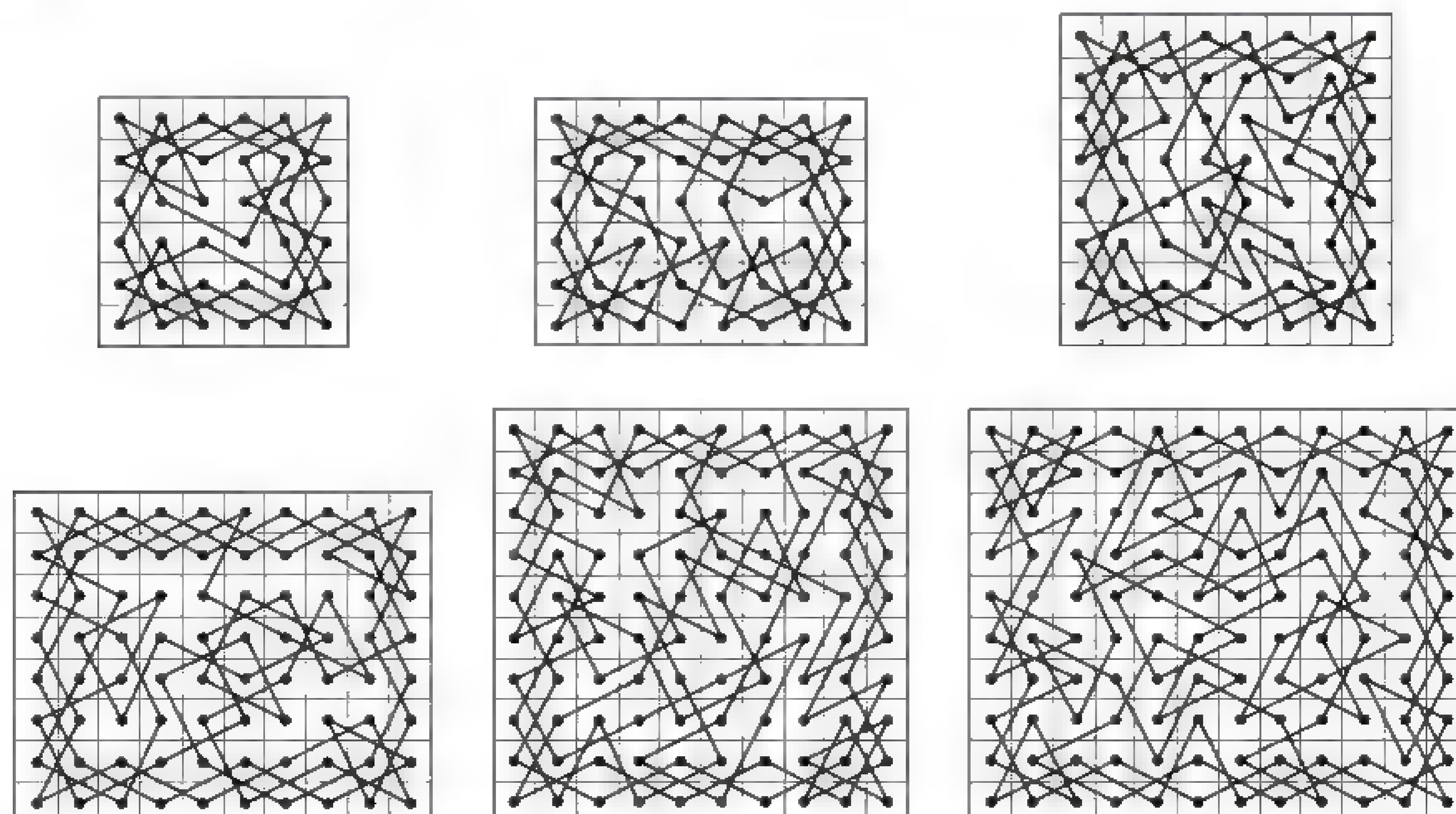


图 2-15 在棋盘上画出的结构化 Hamilton 回路

对于 $m,n \geq 12$ 的情形，采用分治策略。

分割步：

将棋盘尽可能平均地分割成 4 块。当 $m,n=4k$ 时，分割为 2 个 $2k$；当 $m,n=4k+2$ 时，分割为 1 个 $2k$ 和 1 个 $2k+2$。

合并步：

4 个子棋盘拼接后的结构如图 2-16 所示。

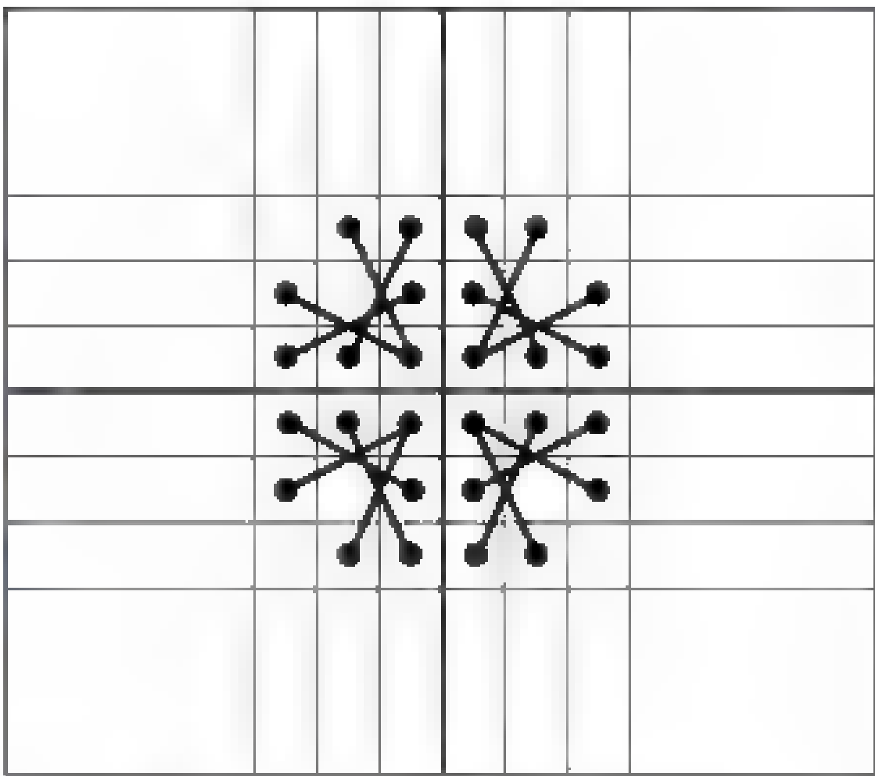


图 2-16　子棋盘的拼接

分别删除 4 个子棋盘中的结构化边 A,B,C,D，添入新边 E,F,G,H，构成整个棋盘的结构化 Hamilton 回路，如图 2-17 所示。

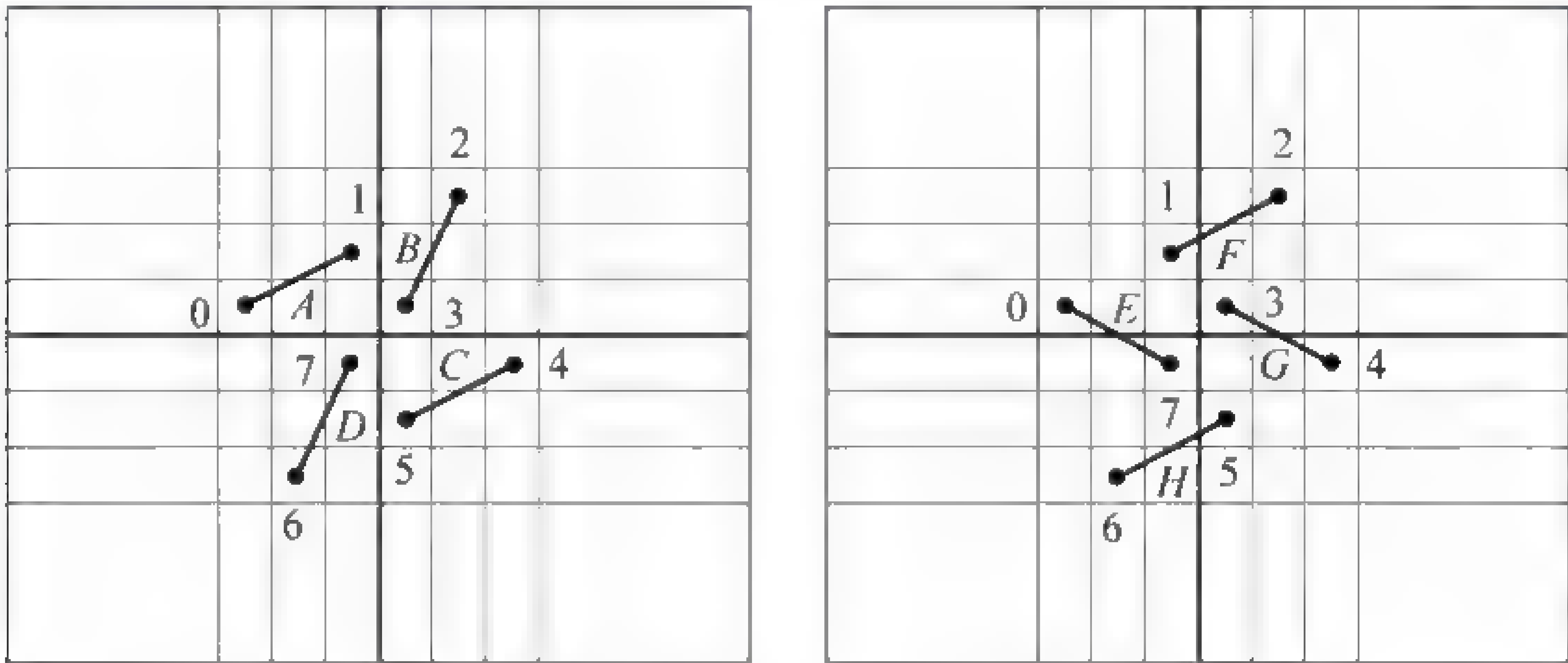


图 2-17　子棋盘中 Hamilton 回路的合并

上述分治法得到的 Hamilton 回路显然仍是结构化 Hamilton 回路。

图 2-18 是用上述分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路。

1	238	17	242	3	6	31	244	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
18	241	2	7	30	243	248	5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
237	256	239	16	27	4	245	32	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
240	19	8	29	10	247	26	249	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
255	236	11	22	15	28	33	246	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
12	233	20	9	228	23	250	25	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
235	254	231	14	21	252	229	34	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
232	13	234	253	230	227	24	251	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
175	220	191	224	177	180	205	226	115	134	99	130	113	110	149	128
192	223	176	181	204	225	166	179	162	131	114	109	150	129	124	111
219	174	221	190	201	178	163	206	135	116	133	100	153	112	127	148
222	193	182	203	184	165	200	167	132	161	108	151	106	125	154	123
173	218	185	196	189	202	207	164	117	136	105	158	101	152	147	126
186	215	194	183	210	197	168	199	104	139	160	107	144	157	122	155
217	172	213	188	195	170	211	208	137	118	141	102	159	120	143	146
214	187	216	171	212	209	198	169	140	103	138	119	142	145	156	121

图 2-18　用分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路

2) 算法复杂性

设用上述分治法计算 $n \times n$ 棋盘上的 Hamilton 回路所需计算时间为 $T(n)$，则 $T(n)$ 满

足如下递归式

$$T(n)=\begin{cases}O(1) & n<12\\ 4T(n/2)+O(1) & n\geqslant 12\end{cases}$$

解此递归式可得 $T(n)=O(n^2)$。

由此可见,上述算法是一个最优算法。

3) 算法实现

用一个类 Knight 实现算法。

```
public class Knight
{
    public static int m,n;
    public static grid b66[],b68[],b86[],b88[],b810[],b108[],
                    b1010[],b1012[],b1210[],link[][];
}
```

其中,grid 是表示整数对的类,m 和 n 分别表示棋盘的行数和列数,二维数组 link 用来表示 Hamilton 回路。

```
public class grid
{
    int x,y;
    grid(){x=0;y=0;}
}
```

b66,b68,b86,b88,b810,b108,b1010,b1012,b1210 分别表示 6×6,6×8,8×6,8×8,8×10,10×8,10×10,10×12,12×10 棋盘上的结构化 Hamilton 回路。

init 读入基础数据,初始化各数组。

```
public static void init(int mm,int nn)
{
    m=mm;n=nn;
    b66=new grid[36];b68=new grid[48];b86=new grid[48];
    b88=new grid[64];b810=new grid[80];b108=new grid[80];
    b1010=new grid[100];b1012=new grid[120];b1210=new grid[120];
    link=new grid[m][n];
    int [][]a=new int[10][12];
    for(int i=0;i<36;i++)b66[i]=new grid();
    for(int i=0;i<48;i++)b68[i]=new grid();
    for(int i=0;i<48;i++)b86[i]=new grid();
    for(int i=0;i<64;i++)b88[i]=new grid();
    for(int i=0;i<80;i++)b810[i]=new grid();
    for(int i=0;i<80;i++)b108[i]=new grid();
    for(int i=0;i<100;i++)b1010[i]=new grid();
    for(int i=0;i<120;i++)b1012[i]=new grid();
    for(int i=0;i<120;i++)b1210[i]=new grid();
    for(int i=0;i<m;i++)
        for(int j=0;j<n;j++) link[i][j]=new grid();
```

```
        ReadStream keyboard=new ReadStream();
        for(int i=0;i<6;i++)
            for(int j=0;j<6;j++) a[i][j]=keyboard.readInt();
        step(6,6,a,b66);
        for(int i=0;i<6;i++)
            for(int j=0;j<8;j++) a[i][j]=keyboard.readInt();
        step(6,8,a,b68);step(8,6,a,b86);
        for(int i=0;i<8;i++)
            for(int j=0;j<8;j++) a[i][j]=keyboard.readInt();
        step(8,8,a,b88);
        for(int i=0;i<8;i++)
            for(int j=0;j<10;j++) a[i][j]=keyboard.readInt();
        step(8,10,a,b810);step(10,8,a,b108);
        for(int i=0;i<10;i++)
            for(int j=0;j<10;j++) a[i][j]=keyboard.readInt();
        step(10,10,a,b1010);
        for(int i=0;i<10;i++)
            for(int j=0;j<12;j++) a[i][j]=keyboard.readInt();
        step(10,12,a,b1012);step(12,10,a,b1210);
    }
```

其中,step 用于将读入的基础棋盘的 Hamilton 回路转化为网格数据。

```
    public static void step(int m,int n,int [][]a,grid b[])
    {
        int i,j,k=m*n;
        if(m<n)
        {
            for (i=0; i<m; i++)
                for (j=0;j<n;j++)
                {
                    int p=a[i][j]-1;
                    b[p].x=i;b[p].y=j;
                }
        }
        else
        {
            for (i=0; i<m; i++)
                for (j=0;j<n;j++)
                {
                    int p=a[j][i]-1;
                    b[p].x=i;b[p].y=j;
                }
        }
    }
```

分治法的主体由算法 comp 给出。

```
    public static boolean comp(int mm,int nn,int offx,int offy)
```

```
{
    int mm1,mm2,nn1,nn2;
    int [ ]x=new int[8];
    int [ ]y=new int[8];
    int []p=new int[8];
    if(odd(mm) || odd(nn) || mm-nn>2 || nn-mm>2 || mm<6 || nn<6)return true;
    if(mm<12 || nn<12){base(mm,nn,offx,offy);return false;}
    mm1=mm/2;
    if(mm%4>0)mm1--;
    mm2=mm-mm1;
    nn1=nn/2;
    if(nn%4>0)nn1--;
    nn2=nn-nn1;
    comp(mm1,nn1,offx,offy);
    comp(mm1,nn2,offx,offy+nn1);
    comp(mm2,nn1,offx+mm1,offy);
    comp(mm2,nn2,offx+mm1,offy+nn1);
    x[0]=offx+mm1-1;y[0]=offy+nn1-3;
    x[1]=x[0]-1;y[1]=y[0]+2;
    x[2]=x[1]-1;y[2]=y[1]+2;
    x[3]=x[2]+2;y[3]=y[2]-1;
    x[4]=x[3]+1;y[4]=y[3]+2;
    x[5]=x[4]+1;y[5]=y[4]-2;
    x[6]=x[5]+1;y[6]=y[5]-2;
    x[7]=x[6]-2;y[7]=y[6]+1;
    for(int i=0;i<8;i++)p[i]=pos(x[i],y[i],n);
    for(int i=1;i<8;i+=2){
        int j1=(i+1)%8,j2=(i+2)%8;
        if(link[x[i]][y[i]]. x==p[i-1])link[x[i]][y[i]]. x=p[j1];
        else link[x[i]][y[i]]. y=p[j1];
        if(link[x[j1]][y[j1]]. x==p[j2])link[x[j1]][y[j1]]. x=p[i];
        else link[x[j1]][y[j1]]. y=p[i];
    }
    return false;
}
```

其中,base 根据基础解构造子棋盘的结构化 Hamilton 回路。

```
public static void base(int mm,int nn,int offx,int offy)
{
    if(mm==6 && nn==6)build(mm,nn,offx,offy,n,b66);
    if(mm==6 && nn==8)build(mm,nn,offx,offy,n,b68);
    if(mm==8 && nn==6)build(mm,nn,offx,offy,n,b86);
    if(mm==8 && nn==8)build(mm,nn,offx,offy,n,b88);
    if(mm==8 && nn==10)build(mm,nn,offx,offy,n,b810);
    if(mm==10 && nn==8)build(mm,nn,offx,offy,n,b108);
```

```
        if(mm==10 && nn==10)build(mm,nn,offx,offy,n,b1010);
        if(mm==10 && nn==12)build(mm,nn,offx,offy,n,b1012);
        if(mm==12 && nn==10)build(mm,nn,offx,offy,n,b1210);
    }
```

其实质性的构造由算法 build 完成。

```
    public static void build(int m,int n,int offx,int offy,int col,grid []b)
    {
        int i,p,q,k=m * n;
        for(i=0;i<k;i++)
        {
            int x1=offx+b[i].x,y1=offy+b[i].y,
                x2=offx+b[(i+1)%k].x,y2=offy+b[(i+1)%k].y;
            p=pos(x1,y1,col);q=pos(x2,y2,col);
            link[x1][y1].x=q;link[x2][y2].y=p;
        }
    }
```

其中，pos 用于计算棋盘方格的编号。棋盘方格各行从上到下，各列从左到右依次编号为 0，1，…，$mn-1$。

```
    public static int pos(int x,int y,int col)
    {
        return col * x+y;
    }
```

最后，由 out 按照要求输出计算出的结构化 Hamilton 回路。

```
    public static void out()
    {
        int i,j,k,x,y,p;
        int [][]a=new int[m][n];
        if(comp(m,n,0,0))return;
        for(i=0;i<m;i++)
            for(j=0;j<n;j++)a[i][j]=0;
        i=0;j=0;k=2;a[0][0]=1;
        System.out.print("(0,0)");
        for(p=1;p<m * n;p++)
        {
            x=link[i][j].x;y=link[i][j].y;
            i=x/n;j=x%n;
            if(a[i][j]>0){i=y/n;j=y%n;}
            a[i][j]=k++;
            System.out.print("("+i+","+j+")");
            if((k-1)%n==0)System.out.println();
        }
        System.out.println();
```

```
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)System.out.print(a[i][j]+" ");
        System.out.println();
    }
}
```

算法实现题 2-5 半数集问题

★ 问题描述

给定一个自然数 n,由 n 开始可以依次产生半数集 set(n)中的数如下:

(1) $n\in$ set(n)。

(2) 在 n 的左边加上一个自然数,但该自然数不能超过最近添加的数的一半。

(3) 按此规则进行处理,直到不能再添加自然数为止。

例如,set(6)={6,16,26,126,36,136}。半数集 set(6)中有 6 个元素。

注意,半数集是多重集。

★ 算法设计

对于给定的自然数 n,计算半数集 set(n)中的元素个数。

★ 数据输入

输入数据由文件名为 input.txt 的文本文件提供。

每个文件只有 1 行,给出整数 n(其中 $0<n<1000$)。

★ 结果输出

将计算结果输出到文件 output.txt 中。输出文件只有 1 行,给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

分析与解答:

设 set(n)中的元素个数为 $f(n)$,则显然有 $f(n)=1+\sum_{i=1}^{n/2} f(i)$。

据此可设计求 $f(n)$的递归算法如下:

```
public static int comp(int n)
{
    int ans=1;
    if (n>1)
    for (int i=1;i<=n/2;i++) ans+=comp(i);
    return ans;
}
```

上述算法中显然有很多的重复子问题计算。用数组存储已计算过的结果,避免重复计算,可明显改进算法的效率。改进后的算法如下:

```
public static int comp(int n)
```

```
{
    int ans=1;
    if (a[n]>0)return a[n];
    for (int i=1;i<=n/2;i++) ans+=comp(i);
    a[n]=ans;
    return ans;
}
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    a=new int [Size+1];
    while (n>0)
    {
        for(int i=0;i<=n;i++)a[i]=0;
        a[1]=1;
        System.out.println(comp(n));
        n=keyboard.readInt();
    }
}
```

算法实现题 2-6　半数单集问题

★ 问题描述

给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

(1) $n \in$ set(n)。

(2) 在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半。

(3) 按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6,16,26,126,36,136}。半数集 set(6)中有 6 个元素。

注意，半数集不是多重集。集合中已经有的元素不再添加到集合中。

★ 算法设计

对于给定的自然数 n，计算半数集 set(n)中的元素个数。

★ 数据输入

输入数据由文件名为 input.txt 的文本文件提供。

每个文件只有 1 行，给出整数 n(其中 $0<n<201$)。

★ 结果输出

将计算结果输出到文件 output.txt 中。输出文件只有 1 行，给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

分析与解答：

此题与算法实现题 2-5 类似。主要区别在于此题的半数集为单集，不允许重复元素。

因此在计算时应剔除重复元素。注意到题中条件 $0<n<201$，蕴涵 $0<n/2\leqslant 100$。因此，在计算时，可能产生重复的元素是 2 位数。一个 2 位数 x 重复产生的条件是，在 1 位数 y $x\%10$ 的半数集中已产生 x，因此，$x/10\leqslant y/2$，或等价地 $2(x/10)\leqslant x\%10$。在前面的算法中，加入剔除重复元素的语句即可。

```
public static long comp(int n)
{
    long ans=1;
    if (a[n]>0)return a[n];
    for (int i=1;i<=n/2;i++)
    {
        ans+=comp(i);
        if ((i>10)&&(2*(i/10)<=i%10)) ans-=a[i/10];
    }
    a[n]=ans;
    return ans;
}
```

如果不利用题中对于 n 的范围限制，则应考虑一般情况，即有 $\lg n$ 位数字的情况。

此题还可用散列表或数字检索树等数据结构方法来实现。

算法实现题 2-7　士兵站队问题

★ 问题描述

在已划分成网格的操场上，n 个士兵散乱地站在网格点上。网格点由整数坐标 (x,y) 表示。士兵们可以沿网格边上、下、左、右移动 1 步，但在同一时刻任一网格点上只能有 1 名士兵。按照军官的命令，士兵们要整齐地列成 1 个水平队列，即排列成 (x,y)，$(x+1,y)$，…，$(x+n-1,y)$。如何选择 x 和 y 的值才能使士兵们以最少的总移动步数排成 1 列。

★ 算法设计

计算使所有士兵排成 1 行需要的最少移动步数。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是士兵数 n，$1\leqslant n\leqslant 10\,000$。接下来 n 行是士兵的初始位置，每行两个整数 x 和 y，其中，$-10\,000\leqslant x,y\leqslant 10\,000$。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是士兵排成 1 行需要的最少移动步数。

输入文件示例	输出文件示例
input.txt	output.txt
5	8
1 2	
2 2	
1 3	

3 −2
3 3

分析与解答：

与习题 2 30 解法相同。

算法实现题 2-8 有重复元素的排列问题

★ 问题描述

设 $R=\{r_1,r_2,\cdots,r_n\}$是要进行排列的 n 个元素，其中的元素 $r_1,r_2,\cdots,r_n$ 可能相同。试设计一个算法，列出 R 的所有不同排列。

★ 算法设计

给定 n 以及待排列的 n 个元素，计算出这 n 个元素的所有不同排列。

★ 数据输入

由文件 input. txt 提供输入数据。文件的第 1 行是元素个数 n，$1\leqslant n\leqslant 500$。第 2 行是待排列的 n 个元素。

★ 结果输出

将计算出的 n 个元素的所有不同排列输出到文件 output. txt 中。文件最后 1 行中的数是排列总数。

输入文件示例	输出文件示例
input. txt	output. txt
4	aacc
aacc	acac
	acca
	caac
	caca
	ccaa
	6

分析与解答：

与主教材中的算法 perm 类似。主要区别是对重复元素的处理方式不同。

```
public static void perm(char [] list, int k, int m)
{
    if (k==m)
    {
        ans ++;
        for (int i=0; i <= m; i ++) System.out.print(list[i]);
        System.out.println();
    }
    else
        for (int i=k; i <= m; i ++)
            if(ok(list,k,i))
            {
```

```
                MyMath.swap(list, k, i);
                perm(list, k+1, m);
                MyMath.swap(list, k, i);
```

函数 ok 用于判别重复元素。

```
public static boolean ok(char []list,int k,int i)
{
    if(i>k) for(int t=k;t<i;t++)if(list[t]==list[i])return false;
    return true;
}
```

算法实现题 2-9　排列的字典序问题

★ 问题描述

n 个元素 $\{1,2,\cdots,n\}$ 有 $n!$ 个不同的排列。将这 $n!$ 个排列按字典序排列,并编号为 0,1,…,$n!-1$。每个排列的编号为其字典序值。例如,当 $n=3$ 时,6 个不同排列的字典序值如表 2-4 所示。

表 2-4　$n=3$ 时 6 个不同排列的字典序值

字典序值	0	1	2	3	4	5
排列	123	132	213	231	312	321

★ 算法设计

给定 n 以及 n 个元素 $\{1,2,\cdots,n\}$ 的一个排列,计算出这个排列的字典序值,以及按字典序排列的下一个排列。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n。第 2 行是 n 个元素 $\{1,2,\cdots,n\}$ 的一个排列。

★ 结果输出

将计算出的排列的字典序值和按字典序排列的下一个排列输出到文件 output.txt 中。文件的第 1 行是字典序值,第 2 行是按字典序排列的下一个排列。

输入文件示例	输出文件示例
input.txt	output.txt
8	8227
2 6 4 5 8 1 7 3	2 6 4 5 8 3 1 7

分析与解答:

1) 由排列计算字典序值

设给定的 $\{1,2,\cdots,n\}$ 的排列为 π,其字典序值为 $\text{rank}(\pi,n)$。按字典序的定义显然有

$$(\pi[1]-1)(n-1)! \leqslant \text{rank}(\pi,n) \leqslant \pi[1](n-1)!-1$$

设 r 是 π 在以 $\pi[1]$ 开头的所有排列中的序号，则 r 也是 $[\pi[2],\cdots,\pi[n]]$ 作为集合 $\{1,2,\cdots,n\}/\{\pi[1]\}$ 中排列的字典序值。如果将 $[\pi[2],\cdots,\pi[n]]$ 中每个大于 $\pi[1]$ 的元素都减 1，则得到集合 $\{1,2,\cdots,n-1\}$ 的一个排列 π'，其字典序值也是 r。由此得到如下计算 $\text{rank}(\pi,n)$ 的递归式

$$\text{rank}(\pi,n)=(\pi[1]-1)(n-1)!+\text{rank}(\pi',n-1)$$

其中，

$$\pi'[i]=\begin{cases}\pi[i+1]-1 & \pi[i+1]>\pi[1]\\ \pi[i+1] & \pi[i+1]<\pi[1]\end{cases}$$

初始条件为 $\text{rank}([1],1)=0$。

据此可设计计算 $\text{rank}(\pi,n)$ 的算法 permRank 如下：

```
public static int permRank(int n,int []pi)
{
    int r=0;
    for(int j=1;j<=n;j++) rho[j]=pi[j];
    for(int j=1;j<=n;j++)
    {
        r+=(rho[j]-1) * f[n-j];
        for(int i=j+1;i<=n;i++)if(rho[i]>rho[j])rho[i]--;
    }
    return r;
}
```

其中，$f[j]$ 存储预先计算出的 $j!$ 的值。

2）由字典序值计算排列

对于每个整数 r，$0\leqslant r\leqslant n!-1$，都有唯一的阶层分解 $r=\sum_{i=1}^{n-1}d_i\cdot i!$，$0\leqslant d_i\leqslant i$。

设 $r=\text{rank}(\pi,n)$，则显然有 $\pi[1]=d_{n-1}+1$。

进一步，由 $r'=r-d_{n-1}\cdot(n-1)!=\text{rank}(\pi',n-1)$ 可递归地找到排列 π'。最后令 $\pi[i]=\pi'[i+1]$ 可得到排列 π。

据此可设计计算排列 π 使 $r=\text{rank}(\pi,n)$ 的算法 permUnrank 如下：

```
public static void permUnrank(int n, int r,int []pi)
{
    pi[n]=1;
    for(int j=1;j<n;j++)
    {
        int d=(r%f[j+1])/f[j];
        r-=d * f[j];
        pi[n-j]=d+1;
        for(int i=n-j+1;i<=n;i++)if(pi[i]>d)pi[i]++;
    }
}
```

3) 由排列计算下一个排列

按字典序的定义可设计从一个排列计算下一个排列的算法。对于给定的排列 π，首先找到下标 i，使得 $\pi[i]>\pi[i+1]>\pi[i+2]>\cdots>\pi[n]$；其次找到下标 j，使得 $\pi[i]<\pi[j]$ 且对所有 $j<k\leqslant n$ 有 $\pi[k]<\pi[i]$。然后交换 $\pi[i]$ 和 $\pi[j]$。最后将子排列 $[\pi[i+1],\cdots,\pi[n]]$ 反转。按此思想设计的算法 permSucc 如下：

```
public static boolean permSucc(int n,int []pi)
{
    int i=n-1;
    boolean flag=false;
    pi[0]=0;
    while (pi[i+1]<pi[i])i--;
    if(i>0)
    {
        flag=true;
        int j=n;
        while(pi[j]<pi[i])j--;
        MyMath.swap(pi, i,j);
        for(int h=i+1;h<=n;h++) rho[h]=pi[h];
        for(int h=i+1;h<=n;h++) pi[h]=rho[n+i+1-h];
    }
    return flag;
}
```

算法实现题 2-10　集合划分问题(一)

★ 问题描述

n 个元素的集合{1,2,…,n}可以划分为若干个非空子集。例如，当 $n=4$ 时，集合{1,2,3,4}可以划分为如下15个不同的非空子集。

{{1},{2},{3},{4}}

{{1,2},{3},{4}}

{{1,3},{2},{4}}

{{1,4},{2},{3}}

{{2,3},{1},{4}}

{{2,4},{1},{3}}

{{3,4},{1},{2}}

{{1,2},{3,4}}

{{1,3},{2,4}}

{{1,4},{2,3}}

{{1,2,3},{4}}

{{1,2,4},{3}}

{{1,3,4},{2}}

{{2,3,4},{1}}

{{1,2,3,4}}

★ 算法设计

给定正整数 n，计算出 n 个元素的集合{1,2,…,n}可以划分为多少个不同的非空子集。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是元素个数 n。

★ 结果输出

将计算出的不同的非空子集数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
5	52

分析与解答：

所求的是 Bell 数，满足如下递归式

$$B(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} B(i), \quad B(0) = 1。$$

算法实现题 2-11　集合划分问题(二)

★ 问题描述

n 个元素的集合{1,2,…,n}可以划分为若干个非空子集。例如，当 $n=4$ 时，集合{1,2,3,4}可以划分为如下15个不同的非空子集：

{{1},{2},{3},{4}}

{{1,2},{3},{4}}

{{1,3},{2},{4}}

{{1,4},{2},{3}}

{{2,3},{1},{4}}

{{2,4},{1},{3}}

{{3,4},{1},{2}}

{{1,2},{3,4}}

{{1,3},{2,4}}

{{1,4},{2,3}}

{{1,2,3},{4}}

{{1,2,4},{3}}

{{1,3,4},{2}}

{{2,3,4},{1}}

{{1,2,3,4}}

其中，集合{{1,2,3,4}}由1个子集组成；集合{{1,2},{3,4}}、{{1,3},{2,4}}、{{1,4},{2,3}}、{{1,2,3},{4}}、{{1,2,4},{3}}、{{1,3,4},{2}}、{{2,3,4},{1}}由2个子集组成；集合{{1,2},{3},{4}}、{{1,3},{2},{4}}、{{1,4},{2},{3}}、{{2,3},{1},{4}}、{{2,4},{1},{3}}、{{3,4},{1},{2}}由3个子集组成；集合{{1},{2},{3},{4}}由4个子集组成。

★ 算法设计

给定正整数 n 和 m，计算出 n 个元素的集合 $\{1,2,\cdots,n\}$ 可以划分为多少个不同的由 m 个非空子集组成的集合。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是元素个数 n 和非空子集数 m。

★ 结果输出

将计算出的不同的由 m 个非空子集组成的集合数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4 3	6

分析与解答：

所求的是第2类 Stirling 数 $S(n,m)$，满足如下递归式

$$S(n,m)=mS(n-1,m)+S(n-1,m-1);\ S(n,n+1)=0,S(n,0)=0,S(0,0)=1$$

关于 Bell 数，显然有 $B(n)=\sum_{m=1}^{n}S(n,m)$。

```
public static void StirlingNumbers2(int m,int n)
{
    int min;
    S[0][0]=1;
    for(int i=1;i<=m;i++) S[i][0]=0;
    for(int i=0;i<m;i++) S[i][i+1]=0;
    for(int i=1;i<=m;i++)
    {
        if(i<n) min=i;else min=n;
        for(int j=1;j<=min;j++) S[i][j]=j*S[i-1][j]+S[i-1][j-1];
    }
}

public static int computeB(int m)
{
    StirlingNumbers2(m,m);
    for(int i=0;i<m;i++) B[i]=0;
    for(int i=1;i<=m;i++)
        for(int j=0;j<=i;j++) B[i-1]+=S[i][j];
    return B[m-1];
}
```

算法实现题 2-12　双色 Hanoi 塔问题

★ 问题描述

设 A、B、C 是3个塔座。开始时，在塔座 A 上有一叠共 n 个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为 $1,2,\cdots,n$，奇数号圆盘着灰色，偶数号圆盘着黑色，如图 2-19 所示。现要求将塔座 A 上的这一叠圆盘移到塔座 B 上，并

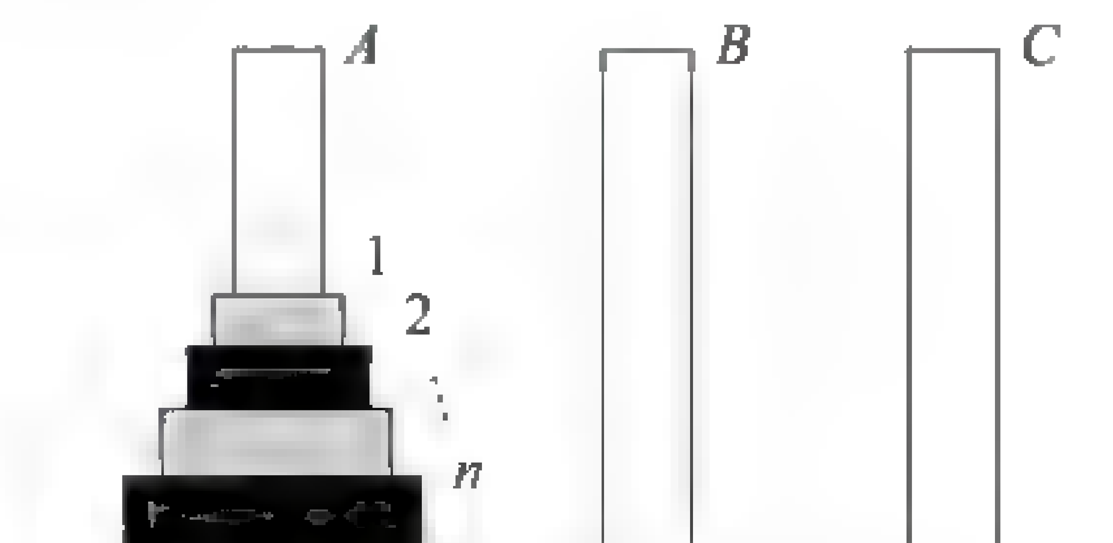


图 2-19　双色 Hanoi 塔

仍按同样顺序叠置。在移动圆盘时应遵守以下规则。

规则(1)：每次只能移动1个圆盘。

规则(2)：任何时刻都不允许将较大的圆盘压在较小的圆盘之上。

规则(3)：任何时刻都不允许将同色圆盘叠在一起。

规则(4)：在满足移动规则(1)～(3)的前提下，可将圆盘移至 A,B,C 中任一塔座上。

试设计一个算法，用最少的移动次数将塔座 A 上的 n 个圆盘移到塔座 B 上，并仍按同样顺序叠置。

★ 算法设计

对于给定的正整数 n，编程计算最优移动方案。

★ 数据输入

由文件 input.txt 给出输入数据。第1行是给定的正整数 n。

★ 结果输出

将计算出的最优移动方案输出到文件 output.txt。文件的每一行由一个正整数 k 和两个字符 c_1 和 c_2 组成，表示将第 k 个圆盘从塔座 c_1 移到塔座 c_2 上。

输入文件示例	输出文件示例
input.txt	output.txt
3	1 A B
	2 A C
	1 B C
	3 A B
	1 C A
	2 C B
	1 A B

分析与解答：

可用主教材中的标准 Hanoi 塔算法。问题是要证明标准 Hanoi 塔算法不违反规则(3)。

用数学归纳法。

设 hanoi(n, A,B,C)将塔座 A 上的 n 个圆盘，以塔座 C 为辅助塔座，移到目的塔座 B 上的标准 Hanoi 塔算法。

归纳假设：当圆盘个数小于 n 时，hanoi(n, A,B,C) 不违反规则(3)，且在移动过程中，目的塔座 B 上最底圆盘的编号与 n 具有相同奇偶性，辅助塔座 C 上最底圆盘的编号与 n 具有不同奇偶性。

当圆盘个数为 n 时，标准 Hanoi 塔算法 hanoi(n, A,B,C)由以下3个步骤完成：

① hanoi($n-1$, A,C,B)。

② move(A,B)。

③ hanoi($n-1$, C,B,A)。

按归纳假设，步骤①不违反规则(3)，且在移动过程中，塔座 C 上最底圆盘的编号与 $n-1$ 具有相同奇偶性，塔座 B 上最底圆盘的编号与 $n-1$ 具有不同奇偶性，从而塔座 B 上最底圆盘的编号与 n 具有相同奇偶性，塔座 C 上最底圆盘的编号与 n 具有不同奇偶性。

步骤②也不违反规则(3)，且塔座 B 上最底圆盘的编号与 n 相同。

按归纳假设，步骤③不违反规则(3)，且在移动过程中，塔座 B 上倒数第 2 个圆盘的编号与 $n-1$ 具有相同奇偶性，塔座 A 上最底圆盘的编号与 $n-1$ 具有不同奇偶性，从而塔座 B 上倒数第 2 个圆盘的编号与 n 具有不同奇偶性，塔座 A 上最底圆盘的编号与 n 具有相同奇偶性。因此在移动过程中，塔座 B 上圆盘不违反规则(3)，而且塔座 B 上最底圆盘的编号与 n 具有相同奇偶性，塔座 C 上最底圆盘的编号与 n 具有不同奇偶性。

由数学归纳法即知，hanoi(n, A,B,C) 不违反规则(3)。

算法实现题 2-13　标准二维表问题

★ 问题描述

设 n 是一个正整数。$2\times n$ 的标准二维表是由正整数 $1,2,\cdots,2n$ 组成的 $2\times n$ 数组，该数组的每行从左到右递增，每列从上到下递增。$2\times n$ 的标准二维表全体记为 Tab(n)。例如，当 $n=3$ 时 Tab(3)如下：

1	2	3
4	5	6

1	2	4
3	5	6

1	2	5
3	4	6

1	3	4
2	5	6

1	3	5
2	4	6

★ 算法设计

给定正整数 n，计算 Tab(n)中 $2\times n$ 的标准二维表的个数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有一个正整数 n。

★ 结果输出

将计算出的 Tab(n)中 $2\times n$ 的标准二维表的个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	5

分析与解答：

Catalan 数。

算法实现题 2-14　整数因子分解问题

★ 问题描述

大于 1 的正整数 n 可以分解为 $n=x_1\times x_2\times\cdots\times x_m$。

例如，当 $n=12$ 时，共有如下 8 种不同的分解式：

$$12=12$$
$$12=6\times 2$$
$$12=4\times 3$$
$$12=3\times 4$$
$$12=3\times 2\times 2$$
$$12=2\times 6$$
$$12=2\times 3\times 2$$
$$12=2\times 2\times 3$$

★ **算法设计**

对于给定的正整数 n，计算 n 共有多少种不同的分解式。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有一个正整数 n(其中 $1 \leqslant n \leqslant 2\ 000\ 000\ 000$)。

★ **结果输出**

将计算出的不同的分解式种数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
12	8

分析与解答：

对 n 的每个因子递归搜索如下：

```
public static void solve(int n)
{
    if (n==1) total++;
    else for (int i=2; i<=n; i++) if (n%i==0) solve(n/i);
}
```

此题可用动态规划求解。

算法实现题 2-15　有向直线 2 中值问题

★ **问题描述**

给定一条有向直线 L 以及 L 上的 $n+1$ 个点 $x_0 < x_1 < \cdots < x_n$。有向直线 L 上的每个点 x_i 都有一个权 $w(x_i)$；每条有向边 (x_i, x_{i-1}) 也都有一个非负边长 $d(x_i, x_{i-1})$。有向直线 L 上的每个点 x_i 可以看作客户，其服务需求量为 $w(x_i)$。每条边 (x_i, x_{i-1}) 的边长 $d(x_i, x_{i-1})$ 可以看作运输费用。如果在点 x_i 处未设置服务机构，则将点 x_i 处的服务需求沿有向边转移到点 x_j 处，服务机构需付出的服务转移费用为 $w(x_i) \times d(x_i, x_j)$。在点 x_0 处已设置了服务机构，现在要在直线 L 上增设两处服务机构，使得整体服务转移费用最小。

★ **算法设计**

对于给定的有向直线 L，计算在直线 L 上增设两处服务机构的最小服务转移费用。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有一个正整数 n，表示有向直线 L 上除了点 x_0 外还有 n 个点 $x_0 < x_1 < \cdots < x_n$。接下来的 n 行中，每行有两个整数。第 $i+1$ 行的两个整数分别表示 $w(x_{n-i-1})$ 和 $d(x_{n-i-1}, x_{n-i-2})$。

★ **结果输出**

将计算的最小服务转移费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
9	26
1 2	

```
2 1
3 3
1 1
3 2
1 6
2 1
1 2
1 1
```

分析与解答:

设 cost(i,j)是在 x_i 和 x_j 处(其中 $i<j$)设置服务机构的费用,则问题是求 $i<j$ 的最优位置,使 cost(i,j)达到最小。

假设在位置 i 已取定,j 的最优位置为 opt(i),则 opt(i)具有如下性质:

对任何 $i<j$,总有 opt$(i)<$opt(j)。

利用此性质可设计如下分治算法:

首先计算 $a=$opt$(n/2)$。由上述性质知,当 $i<n/2$ 时,opt$(i)<a$;当 $i>n/2$ 时,opt$(i)>a$。

据此设计分治算法 comp(minp1,maxp1,minp2,maxp2),找出 $x_i \in \{x_{\min p1},\cdots,x_{\max p1}\}$,$x_j \in \{x_{\min p2},\cdots,x_{\max p2}\}$ 使 cost(i,j)达到最小。

readin 读入初始数据并进行预处理计算。

```
public static void readin()
{
    int d,w;
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    dist[1]=0;
    wt[1]=keyboard.readInt();
    dist[2]=keyboard.readInt();
    tot=wt[1] * dist[2];
    for (int i=2;i<=n;i++)
    {
        w=keyboard.readInt();
        d=keyboard.readInt();
        wt[i]=wt[i-1]+w;
        dist[i+1]=dist[i]+d;
        tot=tot+wt[i] * d;
    }
}
```

getcost 计算 cost(i,j)的值。

```
public static int getcost(int i,int j)
{
```

```
    if(i>j)return 0;
    else return tot-wt[i]*(dist[j]-dist[i])-wt[j]*(dist[n+1]-dist[j]);
}
```

comp 递归计算最优值如下：

```
public static void comp(int minp1,int maxp1,int minp2,int maxp2)
{
    int i,j,cost,opt,optcost;
    if(minp1>=minp2) minp2=minp1+1;
    if(maxp1>=maxp2) maxp1=maxp2-1;
    if((minp1>maxp1) || (minp2>maxp2)) return;
    // 计算 opt((minp2+maxp2)/2)
    optcost=MAXCOST+1;opt=0;
    j=(minp2+maxp2)/2;
    for (i=minp1;i<=min(maxp1,j-1);i++)
    {
        cost=getcost(i,j);
        if(cost<optcost)
        {
            opt=i;
            optcost=cost;
        }
    }
    if(optcost<mincost) mincost=optcost;
    comp(minp1,opt,minp2,(minp2+maxp2)/2-1);
    comp(opt,maxp1,(minp2+maxp2)/2+1,maxp2);
}
```

comp(1,n-1,2,n)完成整个计算。

```
public static void main(String [] args)
{
    readin();
    mincost=MAXCOST+1;
    comp(1,n-1,2,n);
    System.out.println(mincost);
}
```

算法所需的计算时间为 $O(n\log n)$。

第 3 章 动态规划

习题 3-1　最长单调递增子序列

设计一个 $O(n^2)$ 时间的算法，找出由 n 个数组成的序列的最长单调递增子序列。

分析与解答：

用数组 $b[0:n-1]$ 记录以 $a[i]$，$0 \leqslant i < n$ 为结尾元素的最长递增子序列的长度。序列 a 的最长递增子序列的长度为 $\max\limits_{0 \leqslant i < n}\{b[i]\}$。易知，$b[i]$ 满足最优子结构性质，可以递归地定义为

$$b[0]=1, \quad b[i]=\max_{\substack{0 \leqslant k < i \\ a[k] \leqslant a[i]}}\{b[k]\}+1$$

据此将计算 $b[i]$ 转化为 i 个规模更小的子问题。

按此思想设计的动态规划算法描述如下：

```
public static int LISdyna()
{
    int i,j,k;
    for (i=1,b[0]=1;i<n;i++)
    {
        for (j=0,k=0;j<i;j++) if( a[j]<=a[i] && k<b[j] ) k=b[j];
        b[i]=k+1;
    }
    return maxL(n);
}

static int maxL(int n)
{
    int temp=0;
    for (int i=0;i<n;i++) if(b[i]>temp) temp=b[i];
    return temp;
}
```

上述算法 LISdyna 按照递归式计算出 $b[0:n-1]$ 的值，然后由 maxL 计算出序列 a 的最长递增子序列的长度。从算法 LISdyna 的二重循环容易看出，算法所需的计算时间为 $O(n^2)$。

习题 3-2　最长单调递增子序列的 $O(n\log n)$ 算法

将习题 3-1 中算法的计算时间减至 $O(n\log n)$。(提示：一个长度为 i 的候选子序列的最后一个元素至少与一个长度为 $i-1$ 的候选子序列的最后一个元素同样大。通过指向输入序列中元素的指针来维持候选子序列)。

分析与解答：

可用归纳设计策略解此问题。归纳假设是：已知计算序列 $a[0:i-1](i<n)$ 的最长递增子序列的长度的正确算法。归纳的初始情况是平凡的。对于长度为 n 的序列 $a[0:n-1]$，应设法转换为长度小于 n 的序列。

用归纳设计策略解题时，归纳假设对应于算法循环中的循环不变式。本题的循环不变式 P 是：

P：k 是序列 $a[0:i]$ 的最长递增子序列的长度，$0\leqslant i<n$。

容易看出在由 $i-1$ 到 i 的循环中，$a[i]$的值起关键作用。如果 $a[i]$能扩展序列 $a[0:i-1]$的最长递增子序列的长度，则 $k=k+1$，否则 k 不变。设 $a[0:i-1]$的长度为 k 的最长递增子序列的结尾元素是 $a[j](0\leqslant j\leqslant i-1)$，则当 $a[i]\geqslant a[j]$时可以扩展，否则不能扩展。如果序列 $a[0:i-1]$中有多个长度为 k 的最长递增子序列，那么需要存储哪些信息？容易看出，只要存储序列 $a[0:i-1]$中所有长度为 k 的递增子序列中结尾元素的最小值 $b[k]$即可。因此，需要将循环不变式 P 增强为：

P：$0\leqslant i<n$，k 是序列 $a[0:i]$ 的最长递增子序列的长度。

$b[k]$是序列 $a[0:i]$中所有长度为 k 的递增子序列中的最小结尾元素值。

相应的归纳假设也增强为：已知计算序列 $a[0:i-1](i<n)$的最长递增子序列的长度 k 以及序列 $a[0:i-1]$中所有长度为 k 的递增子序列中的最小结尾元素值 $b[k]$的正确算法。

增强归纳假设后，在由 $i-1$ 到 i 的循环中，当 $a[i]\geqslant b[k]$时，$k=k+1$，$b[k]=a[i]$，否则 k 值不变。注意到当 $a[i]\geqslant b[k]$时，k 值增加，$b[k]$的值为 $a[i]$。那么当 $a[i]<b[k]$时，$b[1:k]$的值应该如何改变？如果 $a[i]<b[1]$，则显然应该将 $b[1]$的值改变为 $a[i]$。当 $b[1]\leqslant a[i]\leqslant b[k]$时，注意到数组 b 是有序的，可以用二分搜索算法找到下标 j，使得 $b[j-1]\leqslant a[i]<b[j]$。此时，$b[1:j-1]$和 $b[j+1:k]$的值不变；$b[j]$的值改变为 $a[i]$。

按上述思想设计的算法可实现如下：

```
public static int LIS()
{
    int i=1,k;
    b[1]=a[0];
    for (i=1,k=1;i<n;i++)
    {
        if (a[i]>=b[k]) b[++k]=a[i];
        else b[binary(i,k)]=a[i];
    }
    return k;
}

static int binary(int i, int k)
{
    int h,j;
```

```
        if (a[i]<b[1]) return 1;
        for(h=1, j=k; h!=j-1;)
        {
            if (b[k=(h+j)/2]<=a[i]) h=k;
            else j=k;
        }
        return j;
}
```

binary(i,k)用二分搜索算法在$b[1:k]$中找到下标j,使得$b[j-1]\leq a[i]<b[j]$。算法binary(i,k)所需的计算时间显然为$O(\log k)$。由此可见,在最坏情况下,上述算法所需的计算时间为$O(n\log n)$。

习题 3-3　漂亮打印

给定由n个英文单词组成的一段文章,每个单词的长度(字符个数)依序为$l_1,l_2,\cdots,l_n$。要在一台打印机上将这段文章"漂亮"地打印出来。打印机每行最多可打印M个字符。这里所说的"漂亮"的定义如下:在打印机所打印的每一行中,行首和行尾可不留空格;行中每两个单词之间留一个空格;如果在一行中打印从单词i到单词j的字符,则按打印规则,应在一行中恰好打印$\sum_{k=i}^{j}l_k+j-i$个字符(包括字间空格字符),且不允许将单词打破。多余的空格数为$M-j+i-\sum_{k=i}^{j}l_k$;除文章的最后一行外,希望每行多余的空格数尽可能少。因此,以各行(最后一行除外)的多余空格数的立方和达到最小作为"漂亮"的标准。试用动态规划算法设计一个"漂亮打印"方案,并分析算法的计算复杂性。

分析与解答:

1) 最优子结构性质

在一个最优打印方案中,如果将单词$1\sim k$安排在第1行打印,则显然这个方案对后续的$k+1\sim n$个单词的打印安排是单词$k+1\sim n$的漂亮打印子问题的最优打印方案。

2) 重叠的子问题

在第一行中安排单词$1\sim k$的所有打印方案都要考虑后续单词$k+1\sim n$的打印子问题。因此,在计算中有许多重叠的子问题。

3) 递归计算

由于在一行中不允许将单词打破,故可设$l_i\leq M,1\leq i<n$。为了处理边界情形,定义数组extras和lc为

$$\text{extras}[i][j]=M-j+i-\sum_{k=i}^{j}l_k$$

它是将单词$i\sim j$安排在一行中时行尾的多余空格数。注意,extras$[i][j]$可能为负,即一行不够打印$i\sim j$,有可能将单词打破。

lc$[i][j]$表示将单词$i\sim j$打印在一行上的费用:

$$\text{lc}[i][j]=\begin{cases}\infty & \text{extras}[i][j]<0\\ 0 & j=n,\text{extras}[i][j]\geq 0\\ (\text{extras}[i][j])^3 & \text{其他}\end{cases}$$

设$c[j]$是安排单词$1\sim j$时的最小费用,则所要求的最小费用是$c[n]$。

$c[j],j=1\sim n$ 可递归地计算如下：

$$c[j]=\begin{cases}0 & j=0\\ \min\limits_{1\leqslant i\leqslant j}\{c[i-1]+lc[i][j]\} & j>0\end{cases}$$

算法所需的计算时间为 $O(nM)$。算法所需的空间显然为 $O(n)$。

习题 3-4　整数线性规划问题

考虑下面的整数线性规划问题：

$$\max\sum_{i=1}^{n}c_ix_i$$

$$\sum_{i=1}^{n}a_ix_i\leqslant b$$

x_i 为非负整数，$1\leqslant i\leqslant n$

试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

分析与解答：

该问题是一般情况下的背包问题，具有最优子结构性质。

设所给背包问题的子问题：

$$\max\sum_{k=1}^{i}c_kx_k$$

$$\sum_{k=1}^{i}a_kx_k\leqslant j$$

的最优值为 $m(i,j)$，即 $m(i,j)$ 是背包容量为 j，可选择物品为 $1,2,\cdots,i$ 时背包问题的最优值。由背包问题的最优子结构性质，可以建立计算 $m(i,j)$ 的递归式如下：

$$m(i,j)=\begin{cases}\max\{m(i-1,j),m(i,j-a_i)+c_i\} & j\geqslant a_i\\ m(i-1,j) & 0\leqslant j<a_i\end{cases}$$

$$m(0,j)=m(i,0)=0;\quad m(i,j)=-\infty,\qquad j<0$$

按此递归式计算出的 $m(n,b)$ 为最优值。算法所需的计算时间为 $O(nb)$。

习题 3-5　二维背包问题

给定 n 种物品和一背包。物品 i 的重量是 w_i，体积是 b_i，其价值为 v_i，背包的容量为 c，容积为 d。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

分析与解答：

该问题是二维 0-1 背包问题。问题的形式化描述是，给定 $c>0$，$d>0$，$w_i>0$，$b_i>0$，$v_i>0$，$1\leqslant i\leqslant n$，要求找出 n 元 0-1 向量（$x_1,x_2,\cdots,x_n$），$x_i\in\{0,1\}$，$1\leqslant i\leqslant n$，使得 $\sum\limits_{i=1}^{n}w_ix_i\leqslant c$，$\sum\limits_{i=1}^{n}b_ix_i\leqslant d$ 而且 $\sum\limits_{i=1}^{n}v_ix_i$ 达到最大。

因此，二维 0-1 背包问题也是一个特殊的整数规划问题：

$$\max\sum_{i=1}^{n}v_ix_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \leqslant c \\ \sum_{i=1}^{n} b_i x_i \leqslant d \\ x_i \in \{0,1\}, \quad 1 \leqslant i \leqslant n \end{cases}$$

容易证明该问题具有最优子结构性质。

设所给二维0-1背包问题的子问题：

$$\max \sum_{t=i}^{n} v_t x_t$$

$$\begin{cases} \sum_{t=i}^{n} w_t x_t \leqslant j \\ \sum_{t=i}^{n} b_t x_t \leqslant k \\ x_t \in \{0,1\}, \quad i \leqslant t \leqslant n \end{cases}$$

的最优值为$m(i,j,k)$，即$m(i,j,k)$是背包容量为j，容积为k，可选择物品为$i,i+1,\cdots,n$时二维0-1背包问题的最优值。由二维0-1背包问题的最优子结构性质，可以建立计算$m(i,j,k)$的递归式如下：

$$m(i,j,k)=\begin{cases} \max\{m(i+1,j),m(i+1,j-w_i,k-b_i)+v_i\} & j\geqslant w_i \text{ and } k\geqslant b_i \\ m(i+1,j) & 0\leqslant j<w_i \text{ or } 0\leqslant k<b_i \end{cases}$$

$$m(n,j,k)=\begin{cases} v_n & j\geqslant w_n \text{ and } k\geqslant b_n \\ 0 & 0\leqslant j<w_n \text{ or } 0\leqslant k<b_n \end{cases}$$

按此递归式计算出的$m(n,c,d)$为最优值。算法所需的计算时间为$O(ncd)$。

习题3-6 Ackermann函数

Ackermann函数$A(m,n)$可递归地定义如下：

$$A(m,n)=\begin{cases} n+1 & m=0 \\ A(m-1,1) & m>0,\ n=0 \\ A(m-1,A(m,n-1)) & m>0,\ n>0 \end{cases}$$

试设计一个计算$A(m,n)$的动态规划算法，该算法只占用$O(m)$空间。(提示：用两个数组val[0:m]和ind[0:m]，使得对任何i有val[i]=A(i,ind[i])。)

分析与解答：

按定义容易写出递归算法如下：

```
public static int ackermann(int m, int n)
{
    if(m==0)return n+1;
    if(n==0)return ackermann(m-1, 1);
    else return ackermann(m-1, ackermann(m, n-1));
}
```

按备忘录方法的思想，可将上述算法修改为如下备忘录算法：

```
public static int ack(int m, int n)
```

```
{
    if(a[m][n]>0) return a[m][n];
    if(m==0) return a[0][n]=n+1;
    if(n==0) return a[m][0]=ack(m-1,1);
    return a[m][n]=ack(m-1,ack(m,n-1));
}
```

另外，还可用消去递归的方法，将上述算法非递归化如下：

```
public static int ackm(int m,int n)
{
    int top=1;
    s[1][1]=m; s[1][2]=n;
    while (top>0)
    {
        m=s[top][1];n=s[top][2];top--;
        if (top==0 && m==0) return n+1;
        if (m==0) s[top][2]=n+1;
        else if(n==0){s[++top][1]=m-1;s[top][2]=1;}
        else{s[++top][1]=m-1;s[++top][1]=m;s[top][2]=n-1;}
    }
    return s[0][1];
}
```

实际上，还有一个稍简单的递归算法如下：

```
public static int ack(int m, int n)
{
    for(int i=m;i>0;i--)
    {
        if (n==0)n=1;
        else n=ack(i,n-1);
    }
    return n+1;
}
```

同样，也可将其改造为备忘录算法。

这些算法都是自顶向下的递归算法。下面考查自底向上的动态规划算法，如图 3-1 所示。

$m\backslash n$	0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12
2	3	5	7	9	11	13	15	17	19	21	23
3	5	13	29	61	125	253	509	1021	2045	4093	8189
4	13	65 533									
5	65 533										

图 3-1 Ackermann 函数 $A(m,n)$值的变化

首先对于较小的 m 和 n 考查 Ackermann 函数 $A(m,n)$的值的变化。

从图3-1容易看出,当 $m=0$ 时,第0行对应于 $A(0,n)$ 的值。当 $n=0$ 时,第0列对应于 $A(m,0)$ 的值,其值恰好是第 $m-1$ 行第1列的值。$A(m,n)$ 的值等于第 $m-1$ 行第 $A(m,n-1)$ 列的值。据此,可从第1行的值依次递推出各行的值。为此,用两个数组 val[0:m] 和 ind[0:m] 分别记录当前第 i 行计算到第 ind[i] 列的值,即已计算到 $A(i,\text{ind}[i])$ 的值,这个值存储于 val[i] 中。当计算到 $A(m,n)$ 时,算法结束。按此思想设计的动态规划算法如下:

```
public static int ack(int m, int n)
{
    int i,val[],ind[];
    if(m==0)return n+1;
    val=new int[m+1];
    ind=new int[m+1];
    for(i=0;i<=m;i++){val[i]=-1;ind[i]=-2;}
    val[0]=1;ind[0]=0;
    while(ind[m]<n)
    {
        val[0]++;ind[0]++;
        for(i=0;i<m;i++)
        {
            if(ind[i]==1 && ind[i+1]<0){val[i+1]=val[0];ind[i+1]=0;}
            if(val[i+1]==ind[i]){val[i+1]=val[0];ind[i+1]++;}
        }
    }
    return val[m];
}
```

算法显然只占用 $O(m)$ 空间。

算法实现题 3-1 独立任务最优调度问题

★ 问题描述

用两台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时需要时间 a_i,若由机器 B 来处理,则需要时间 b_i。由于各作业的特点和机器的性能关系,很可能对于某些 i,有 $a_i \geq b_i$,而对于某些 j,$j \neq i$,有 $a_j < b_j$。既不能将一个作业分开由两台机器处理,也没有一台机器能同时处理两个作业。设计一个动态规划算法,使得这两台机器处理完这 n 个作业的时间最短(从任何一台机器开工到最后一台机器停工的总时间)。研究一个实例:$(a_1,a_2,a_3,a_4,a_5,a_6)=(2,5,7,10,5,2)$;$(b_1,b_2,b_3,b_4,b_5,b_6)=(3,8,4,11,3,4)$。

★ 算法设计

对于给定的两台处理机 A 和 B 处理 n 个作业,找出一个最优调度方案,使两台机器处理完这 n 个作业的时间最短。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是1个正整数 n,表示要处理 n 个作业。接下来的两行中,每行有 n 个正整数,分别表示处理机 A 和 B 处理第 i 个作业需要的处理时间。

★ **结果输出**

将计算出的最短处理时间输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
6	15
2 5 7 10 5 2	
3 8 4 11 3 4	

分析与解答：

(1) 首先计算出 $m=\max\{\max\limits_{1\leqslant i\leqslant n}\{a_i\},\max\limits_{1\leqslant i\leqslant n}\{b_i\}\}$。

(2) 设布尔量 $p(i,j,k)$ 表示前 k 个作业可以在处理机 A 用时不超过 i 且处理机 B 用时不超过 j 时间内完成。用动态规划算法计算 $p(i,j,k)=p(i-a_k,j,k-1)\parallel p(i,j-b_k,k-1)$。

(3) 由(2)的结果可以计算最短处理时间为 $\min\limits_{0\leqslant i\leqslant mn,0\leqslant j\leqslant mn,p(i,j,n)=true}\{\max\{i,j\}\}$。

具体算法实现如下：

read 读入初始数据，并计算 m 的值。

```
public static void read()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();m=0;
    a=new int[n];b=new int[n];
    for(int i=0;i<n;i++){a[i]=keyboard.readInt();if(a[i]>m)m=a[i];}
    for(int i=0;i<n;i++){b[i]=keyboard.readInt();if(b[i]>m)m=b[i];}
    mn=m*n;
    p=new boolean[mn+1][mn+1][n+1];
}
```

dyna 实现动态规划算法如下：

```
public static void dyna()
{
    int i,j,k;
    for(i=0;i<=mn;i++)
        for(j=0;j<=mn;j++)
        {
            p[i][j][0]=true;
            for(k=1;k<=n;k++)p[i][j][k]=false;
        }
    for(k=1;k<=n;k++)
        for(i=0;i<=mn;i++)
            for(j=0;j<=mn;j++)
            {
                if(i-a[k-1]>=0)p[i][j][k]=p[i-a[k-1]][j][k-1];
                if(j-b[k-1]>=0)p[i][j][k]=(p[i][j][k]||p[i][j-b[k-1]][k-1]);
            }
```

```
        for(i=0,opt=mn;i<=mn;i++)
            for(j=0;j<=mn;j++)
                if(p[i][j][n])
                {
                    int tmp=(i>j)? i:j;
                    if(tmp<opt)opt=tmp;
                }
        System.out.println(opt);
    }
```

实现算法的主函数如下：

```
    public static void main(String [] args)
    {
        read();
        dyna();
    }
```

上述算法所需的计算时间显然为$O(m^2n^3)$。

该问题的另一解法见参考文献[1]的第53～54页。

算法实现题3-2　最少硬币问题

★问题描述

设有n种不同面值的硬币，各硬币的面值存于数组$T[1:n]$中。现要用这些面值的硬币来找钱。可以使用的各种面值的硬币个数存于数组Coins[1:n]中。

对任意钱数$0\leqslant m\leqslant 20\,001$，设计一个用最少硬币找钱$m$的方法。

★算法设计

对于给定的$1\leqslant n\leqslant 10$，硬币面值数组T和可以使用的各种面值的硬币个数数组Coins，以及钱数m，$0\leqslant m\leqslant 20\,001$，计算找钱$m$的最少硬币数。

★数据输入

由文件input.txt提供输入数据，文件的第一行中只有1个整数给出n的值，第2行起每行2个数，分别是$T[j]$和Coins[j]。最后1行是要找的钱数m。

★结果输出

将计算出的最少硬币数输出到文件output.txt中。问题无解时输出-1。

输入文件示例	输出文件示例
input.txt	output.txt
3	5
1 3	
2 3	
5 3	
18	

分析与解答：

不妨设$0<T[1]<T[2]<\cdots<T[n]$。

当只用面值$T[1:i]$的硬币时，可找出钱数j的最少硬币个数记为$c[i][j]$。

计算 $c[i][j]$ 的递归表达式为：$c[i][j]=\min\{c[i-1][j],1+c[i][j-T[i]]\}$。

据此设计的算法需要 $O(nL)$ 时间和 $O(L)$ 空间。

算法实现题 3-3　序关系计数问题

★ **问题描述**

用关系 $<$ 和 $=$ 将 3 个数 A、B 和 C 依序排列时有如下 13 种不同的序关系：

$A=B=C$，$A=B<C$，$A<B=C$，$A<B<C$，$A<C<B$，$A=C<B$，$B<A=C$，$B<A<C$，$B<C<A$，$B=C<A$，$C<A=B$，$C<A<B$，$C<B<A$。

将 n 个数$(1\leqslant n\leqslant 50)$依序排列时有多少种序关系。

★ **算法设计**

计算出将 n 个数(其中 $1\leqslant n\leqslant 50$)依序排列时有多少种序关系。

★ **数据输入**

由文件 input.txt 提供输入数据。文件只有 1 行，提供 1 个数 n。

★ **结果输出**

将找到的序关系数输出到文件 output.txt 的第 1 行中。

输入文件示例	输出文件示例
input.txt	output.txt
3	13

分析与解答：

设 $A[i][j]$ 表示用 j 个 $<$ 号连接 i 个数时产生的不同序关系数，并约定当 $j>i-1$ 时 $A[i][j]=0$。

当 $j=0$ 时显然有 $A[i][0]=1$，$i=1\sim n$。

一般地，$A[i][j]=\sum\limits_{k=1}^{i-1}\binom{i}{k}A[i-k][j-1]$。

事实上，$A[i][j]$ 有一个更简捷的递归计算式 $A[i][j]=(j+1)(A[i-1][j-1]+A[i-1][j])$。

这个递归式十分简捷，但不易得到。可以通过以下两个途径来推出这个递归式：①用数学归纳法推导；②用组合分析法推导。

算法实现题 3-4　多重幂计数问题

★ **问题描述**

设给定 n 个变量 $x_1,x_2,\cdots,x_n$。将这些变量依序作底和各层幂，可得 n 重幂如下：

$$x_1^{x_2^{x_3^{\cdot^{\cdot^{\cdot^{x_n}}}}}}$$

这里将上述 n 重幂看作是不确定的，当在其中加入适当的括号后，才能成为一个确定的 n 重幂。不同的加括号方式导致不同的 n 重幂。例如，当 $n=4$ 时，全部 4 重幂有 5 个。

★ **算法设计**

对 n 个变量计算出有多少个不同的 n 重幂。

★ **数据输入**

由文件 input.txt 提供输入数据。文件只有 1 行，提供 1 个数 n。

★ 结果输出

将找到的序关系数输出到文件 output.txt 的第1行中。

输入文件示例	输出文件示例
input.txt	output.txt
4	5

分析与解答:

与主教材中矩阵连乘问题解法类似。

算法实现题 3-5　最小 m 段和问题

★ 问题描述

给定 n 个整数组成的序列,现在要求将序列分割为 m 段,每段子序列中的数在原序列中连续排列。如何分割才能使这 m 段子序列的和的最大值达到最小?

★ 算法设计

给定 n 个整数组成的序列,计算该序列的最优 m 段分割,使 m 段子序列的和的最大值达到最小。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行中有两个正整数 n 和 m。正整数 n 是序列的长度,正整数 m 是分割的段数。第2行中有 n 个整数。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第1行中的数是计算出的 m 段子序列的和的最大值的最小值。

输入文件示例	输出文件示例
input.txt	output.txt
1 1	10
10	

分析与解答:

具体算法实现如下:

```
public static void solve(int n,int m)
{
    int i,j,k,temp,maxt;
    for (i=1; i<=n; i++) f[i][1]=f[i-1][1]+t[i];
    for(j=2;j<=m;j++)
        for(i=j;i<=n;i++)
        {
            for(k=1,temp=Integer.MAX_VALUE;k<i;k++)
            {
                maxt=Math.max(f[i][1]-f[k][1],f[k][j-1]);
                if(temp>maxt)temp=maxt;
            }
            f[i][j]=temp;
        }
```

```
}
```

最优值在 $f[n][m]$中。

```
public static void main(String [ ] args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int m=keyboard.readInt();
    if ((n<m) || (n==0)) {System.out.println(0);return;}
    f=new int[n+1][m+1];
    t=new int[n+1];
    for (int i=1; i<=n; i++) t[i]=keyboard.readInt();
    solve(n,m);
    System.out.println(f[n][m]);
}
```

算法实现题 3-6　石子合并问题

★ 问题描述

在一个圆形操场的四周摆放着 n 堆石子,现要将石子有次序地合并成一堆。规定每次只能选相邻的两堆石子合并成新的一堆,并将新的一堆石子数记为该次合并的得分。试设计一个算法,计算出将 n 堆石子合并成一堆的最小得分和最大得分。

★ 算法设计

对于给定 n 堆石子,计算合并成一堆的最小得分和最大得分。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n,$1\leqslant n\leqslant 100$,表示有 n 堆石子,第 2 行有 n 个数,分别表示每堆石子的个数。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是最小得分,第 2 行中的数是最大得分。

输入文件示例	输出文件示例
input.txt	output.txt
4	43
4 4 5 9	54

分析与解答:

石子合并问题实际上是矩阵连乘积问题的变形。设 n 堆石子从左到右编号为 $1,2,\cdots,n$,每堆石子的石子数分别为 $A[1],A[2],\cdots,A[n]$。石子的合并可以有许多不同的方式,每一种合并方式都对应于 $A[1],A[2],\cdots,A[n]$的一种完全加括号方式。因此,该问题转化为 $A[1],A[2],\cdots,A[n]$的最优加括号方式问题。由于问题是要求合并的最小得分,因此,这里的最优的含义是要求使得分达到最小的完全加括号方式。这与矩阵连乘积的最优性含义完全相同,不同之处在于各自的最优值计算公式。

下面给出具体的算法实现。

init 读入初始数据并初始化。

```
public static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    a=new int[MaxLen];
    m=new int[MaxLen][MaxLen];
    s=new int[MaxLen][MaxLen];
    for (int i=0; i <MaxLen; i++)
    {
        a[i]=0;
        for(int j=0;j<MaxLen;j++)
        {
            m[i][j]=0;s[i][j]=0;
        }
    }
    for (int i=1; i<=n; i++) a[i]=keyboard.readInt();
    for (int i=1; i<=n-1; i++) a[n+i]=a[i];
}
```

算法 circle 解圆排列的石子合并问题。

```
public static int circle(int ss)
{
    n=2*n-1;
    if ((ss==1)) minsum(a);
    else maxsum(a);
    n=(n+1)/2;
    int mm=m[1][n];
    for (int i=2; i<=n; i++)
        if ((ss==1) && (m[i][n+i-1]<mm) || (ss>1) && (m[i][n+i-1]>mm))
            mm=m[i][n+i-1];
    return(mm);
}
```

当 ss=1 时,计算最小得分;当 ss>1 时,计算最大得分。
算法 minsum 解直线排列的最小得分石子合并问题。

```
public static void minsum(int a[])
{
    for (int i=2; i<=n; i++) a[i]=a[i]+a[i-1];
    for (int r=2; r<=n; r++)
        for (int i=1; i<=n-r+1; i++)
        {
            int j=i+r-1,i1=i+1,j1=j;
```

```
            if (s[i][j-1]>i) i1=s[i][j-1];
            if (s[i+1][j]>i) j1=s[i+1][j];
            m[i][j]=m[i][i1-1]+m[i1][j];
            s[i][j]=i1;
            for (int k=j1; k>=i1+1; k--)
            {
                int q=m[i][k-1]+m[k][j];
                if ((q<m[i][j])){m[i][j]=q;s[i][j]=k;}
            }
            m[i][j]=m[i][j]+a[j]-a[i-1];
        }
}
```

算法 maxsum 解直线排列的最大得分石子合并问题。

```
public static void maxsum(int a[])
{
    for (int r=2; r<=n; r++)
        for (int i=1; i<=n-r+1; i++)
        {
            int j=i+r-1;
            if (m[i+1][j]>m[i][j-1]) m[i][j]=m[i+1][j]+a[j]-a[i-1];
            else m[i][j]=m[i][j-1]+a[j]-a[i-1];
        }
}
```

算法实现题 3-7　数字三角形问题

★ 问题描述

给定一个由 n 行数字组成的数字三角形如图 3-2 所示。试设计一个算法，计算出从三角形的顶至底的一条路径，使该路径经过的数字总和最大。

★ 算法设计

对于给定的由 n 行数字组成的数字三角形，计算从三角形的顶至底的路径经过的数字和的最大值。

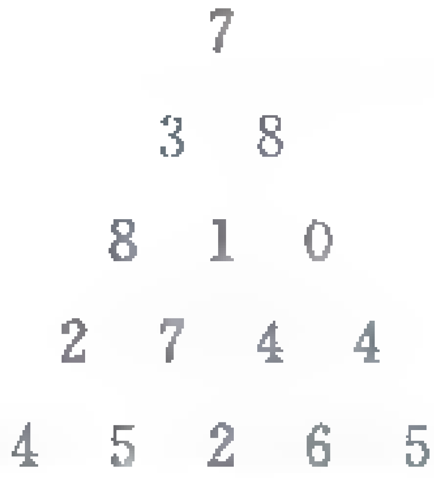


图 3-2　数字三角形

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是数字三角形的行数 n，$1 \leqslant n \leqslant 100$。接下来 n 行是数字三角形各行中的数字，所有数字为 0～99。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的最大值。

输入文件示例	输出文件示例
input.txt	output.txt
5	30
7	

3 8
8 1 0
2 7 4 4
4 5 2 6 5

分析与解答:

以自底向上的方式递归计算。

```
for(int row=N-2; row>=0; row--)
    for (int col=0; col<=row; col++)
        if (triangle[row+1][col]>triangle[row+1][col+1])
            triangle[row][col]+=triangle[row+1][col];
        else triangle[row][col]+=triangle[row+1][col+1];
```

最优值在 triangle[0][0]中。

算法实现题 3-8　乘法表问题

★ 问题描述

定义于字母表 $\Sigma=\{a,b,c\}$ 上的乘法表如表 3-1 所示。

表 3-1　字母表 $\Sigma=\{a,b,c\}$ 上的乘法表

字符	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

依此乘法表,对任一定义于 Σ 上的字符串,适当加括号后得到一个表达式。例如,对于字符串 $x=bbbba$,它的一个加括号表达式为 $(b(bb))(ba)$。依乘法表,该表达式的值为 a。试设计一个动态规划算法,对任一定义于 Σ 上的字符串 $x=x_1x_2\cdots x_n$,计算有多少种不同的加括号方式,使由 x 导出的加括号表达式的值为 a。

★ 算法设计

对于给定的字符串 $x=x_1x_2\cdots x_n$,计算有多少种不同的加括号方式,使由 x 导出的加括号表达式的值为 a。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中给出一个字符串。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的加括号方式数。

输入文件示例	输出文件示例
input.txt	output.txt
bbbba	6

分析与解答:

init 读入初始数据并初始化。

```
public static void init()
{
    ReadStream keyboard=new ReadStream();
    a=keyboard.readString();
```

```
        n=a.length();
        f=new int[n][n][3];
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                for(int k=0;k<3;k++)f[i][j][k]=0;
        for(int i=0;i<n;i++)
            for(int j=0;j<3;j++)
                f[i][i][j]=a.charAt (i)==b.charAt (j)? 1:0;
    }

    public static void dyna()
    {
      for(int k=1;k<n;k++)
        for(int i=0;i<n-k;i++){
          int j=i+k;
          for(int p=i;p<j;p++){
            f[i][j][0]+=f[i][p][2]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*f[p+1][j][2];
            f[i][j][1]+=f[i][p][0]*f[p+1][j][0]+(f[i][p][0]+f[i][p][1])*f[p+1][j][1];
            f[i][j][2]+=f[i][p][1]*f[p+1][j][0]+f[i][p][2]*(f[p+1][j][1]+f[p+1][j][2]);
          }
        }
    }
```

最优值在 $f[0][n-1][0]$中。

算法实现题 3-9　租用游艇问题

★ 问题描述

长江游艇俱乐部在长江上设置了 n 个游艇出租站 $1,2,\cdots,n$。游客可在这些游艇出租站租用游艇，并在下游的任何一个游艇出租站归还游艇。游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i,j)$，$1\leqslant i<j\leqslant n$。试设计一个算法，计算出从游艇出租站 1 到游艇出租站 n 所需的最少租金。

★ 算法设计

对于给定的游艇出租站 i 到游艇出租站 j 之间的租金为 $r(i,j)$，$1\leqslant i<j\leqslant n$，计算从游艇出租站 1 到游艇出租站 n 所需的最少租金。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 $n(n\leqslant 200)$，表示有 n 个游艇出租站，接下来的 $n-1$ 行是 $r(i,j)$，$1\leqslant i<j\leqslant n$。

★ 结果输出

将计算出的从游艇出租站 1 到游艇出租站 n 所需的最少租金输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3	12
5 15	
7	

分析与解答：

init 读入初始数据并初始化。

```
public static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    f=new int[n][n];
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++) f[i][j]=0;
    for(int i=0;i<n-1;i++)
        for(int j=i+1;j<n;j++) f[i][j]=keyboard.readInt();
}

public static void dyna()
{
    for(int k=2;k<n;k++)
        for(int i=0;i<n-k;i++)
        {
            int j=i+k;
            for(int p=i+1;p<j;p++)
            {
                int tmp=f[i][p]+f[p][j];
                if (f[i][j]>tmp) f[i][j]=tmp;
            }
        }
}
```

最优值在 $f[0][n-1]$中。

算法实现题 3-10　汽车加油行驶问题

★ 问题描述

给定一个 $N\times N$ 的方形网格，设其左上角为起点◎，坐标为(1,1)，X 轴向右为正，Y 轴向下为正，每个方格边长为 1。一辆汽车从起点◎出发驶向右下角终点▲，其坐标为(N,N)。在若干个网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则：

(1) 汽车只能沿网格边行驶，装满油后能行驶 K 条网格边。出发时汽车已装满油，在起点与终点处不设油库。

(2) 当汽车行驶经过一条网格边时，若其 X 坐标或 Y 坐标减小，则应付费用 B，否则免付费用。

(3) 汽车在行驶过程中遇油库则应加满油并付加油费用 A。

(4) 在需要时可在网格点处增设油库，并付增设油库费用 C(不含加油费用 A)。

(5) 上述(1)～(4)中的各数 N,K,A,B,C 均为正整数。

★ 算法设计

求汽车从起点出发到达终点的一条所付费用最少的行驶路线。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是 N,K,A,B,C 的值，$2 \leqslant N \leqslant 100$，$2 \leqslant K \leqslant 10$。第 2 行起是一个 $N \times N$ 的 0-1 方阵，每行 N 个值，至 $N+1$ 行结束。方阵的第 i 行第 j 列处的值为 1 表示在网格交叉点 (i,j) 处设置了一个油库，为 0 时表示未设油库。各行相邻的两个数以空格分隔。

★ 结果输出

将找到的最优行驶路线所需的费用，即最小费用输出到文件 output.txt 中。文件的第 1 行中的数是最小费用值。

输入文件示例	输出文件示例
input.txt	output.txt
9 3 2 3 6	12
0 0 0 0 1 0 0 0 0	
0 0 0 1 0 1 1 0 0	
1 0 1 0 0 0 0 1 0	
0 0 0 0 0 1 0 0 1	
1 0 0 1 0 0 1 0 0	
0 1 0 0 0 0 0 1 0	
0 0 0 0 1 0 0 0 1	
1 0 0 1 0 0 0 1 0	
0 1 0 0 0 0 0 0 0	

分析与解答：

用 $f(x,y,0)$ 表示汽车从网格点 $(1,1)$ 行驶至网格点 (x,y) 所需的最少费用；$f(x,y,1)$ 表示汽车行驶至网格点 (x,y) 后还能行驶的网格边数。

建立计算 $f(x,y,0)$ 和 $f(x,y,1)$ 的递归式如下：

$f(1,1,0)=0$，$f(1,1,1)=K$

$f(x,y,0)=f(x,y,0)+A$，$f(x,y,1)=K$，(x,y) 是油库

$f(x,y,0)=f(x,y,0)+C+A$，$f(x,y,1)=K$，(x,y) 非油库且 $f(x,y,1)=0$

$$f(x,y,0)=\min_{0\leqslant i<4}\{f(x+s[i][0],y+s[i][1],0)+s[i][2]\}$$

$$f(x,y,1)=f(x+s[j][0],y+s[j][1],1)-1$$

其中，数组 $s=\{\{-1,0,0\},\{0,-1,0\},\{1,0,B\},\{0,1,B\}\}$。

用备忘录方法递归计算。$f(n,n,0)$ 为最优值。

算法实现题 3-11　圈乘运算问题

★ 问题描述

关于整数的 2 元圈乘运算 $\otimes$ 定义如下：

$(X\otimes Y)$ 等于十进制整数 X 的各位数字之和乘以十进制整数 Y 的最大数字加上 Y 的最小数字。

例如，$(9\otimes 30)=9\times 3+0=27$。

对于给定的十进制整数 X 和 K，由 X 和 $\otimes$ 运算可以组成各种不同的表达式。试设计一

个算法,计算出由 X 和⊙运算组成的值为 K 的表达式最少需用多少个⊙运算。

★ **算法设计**

给定十进制整数 X 和 K(其中 $1 \leqslant X, K \leqslant 10^{20}$)。计算由 X 和⊙运算组成的值为 K 的表达式最少需用多少个⊗运算。

★ **数据输入**

输入数据由文件名为 input.txt 的文本文件提供。

每一行有两个十进制整数 X 和 K。

最后一行是 0 0。

★ **结果输出**

将找到的最少⊗运算个数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3 12	1
0 0	

分析与解答:

(1) ⊗运算一般不满足交换律和结合律。

(2) 最优子结构性质:设⊙运算表达式 $(X \odot Y) = K$ 用了最少的⊙运算,则 X 和 Y 也用了最少的⊗运算。

(3) ⊙运算表达式值的有限性:对于给定的 N,其十进制位数为 $m = \lceil \lg(N+1) \rceil$。易知,当 $m > 1$ 时,⊙运算表达式最大值不超过 $81 \times m + 9$;当 $m = 1$ 时,⊙运算表达式最大值不超过 171。

设对于给定的 N,⊙运算表达式的值域为 $S(N)$,则 $S(N) \subseteq [1:L] \cup \{N\}$,其中,

$$L = 81 \times \max\{2, \lceil \lg(N+1) \rceil\} + 9。$$

因此,对于给定的 N,⊗运算表达式最大值为 $O(\log N)$ 量级。

(4) 从 N 出发,反复用 N 和⊙运算可求得 $S(N)$ 中每一个数。在计算过程中,用动态规划算法求 $S(N)$ 中每一个数所用的最少⊗运算次数。

(5) 数据结构:用数组 num[0:L][0:3]存储 $S(N)$ 中每个数的相关信息。

对于任意正整数 x,用 sum(x),max(x)和 min(x)分别记其各位数字之和、最大数字和最小数字。num[i][0]存储 i 所需的最少⊙运算次数;num[i][1]存储 min(i);num[i][2]存储 max(i);num[i][3]存储 sum(i)。min(N),max(N)和 sum(N)分别存储于 num[0][1],num[0][2]和 num[0][3]中。

(6) 算法实现。

首先由 input 输入 N 和 K。计算 L,当 $K > L$ 时,明显无解。

```
public static int input()
{
    ReadStream keyboard = new ReadStream();
    s1 = keyboard.readString();
    s2 = keyboard.readString();
    if (equals(s1,s2)) {out(0);return 0;}
    len = s1.length();
```

```
    big=81 * len+9;
    if (big<171) big=171;
    int biglen=(int)Math.ceil (Math.log(big)/Math.log(10.0));
    if (s2.length()>biglen) {out(-1);return 0;}
    kk=Integer.parseInt(s2,10);
    if (kk>big) {out(-1);return 0;}
    return 1;
}
```

然后，由 init 初始化数组 num。

```
public static void init()
{
    num=new int[++big][4];
    for (int i=0;i<big;i++){
        num[i][0]=Integer.MAX_VALUE;
        num[i][1]=Integer.MAX_VALUE;
        num[i][2]=0;num[i][3]=0;
    }
    for(int i=0;i<len;i++)      count(ctoi(s1.charAt(i)),0);
    num[0][0]=0;
    for (int i=1;i<big;i++)
    {
        int t=i;
        while(t>0){int j=t%10;t/=10;count(j,i);}
    }
}
```

上述程序中用到 ctoi，count 和 out 如下：

```
public static int ctoi(char x)
{
    return (int)x-48;
}

public static void count(int i,int j)
{
    if (i<num[j][1]) num[j][1]=i;
    if (i>num[j][2]) num[j][2]=i;
    num[j][3]+=i;


public static void out(int x)
{
    if (x>=0) System.out.println(x);
    else System.out.println("No answer!");
}
```

算法 compute 从 N 出发,反复用 N 和⊗运算求 $S(N)$中每一个数。在计算过程中,用动态规划算法求 $S(N)$中每一个数所用的最少⊗运算次数。

```
public static void compute()
{
    boolean flag=true;
    while(flag)
    {
        flag=false;
        for (int i=0;i<big;i++)
            if (num[i][0]<Integer.MAX_VALUE)
                for (int k=0;k<big;k++)
                    if (num[k][0]<Integer.MAX_VALUE)
                    {
                        int j=num[i][3] * num[k][2]+num[k][1];
                        if (num[i][0]+num[k][0]+1<num[j][0])
                        {
                            num[j][0]=num[i][0]+num[k][0]+1;
                            flag=true;
                        }
                    }
    }
    if (num[kk][0]<Integer.MAX_VALUE) out(num[kk][0]);
    else out(-1);
}
```

完成整个计算的主函数 main 如下:

```
public static int kk,len,big;
public static int [][]num;
public static String s1,s2;

public static void main(String [] args)
{
    if(input()>0) init();
    else return;
    compute();
}
```

(7) 算法的计算复杂性。

算法所需空间显然为 $O(\log N)$。

算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的1次 while 循环需要 $O(\log^2 N)$ 的计算时间。最坏情况下需要 $O(\log N)$ 次 while 循环。因此,算法在最坏情况下所需的计算时间为 $O(\log^3 N)$。

(8) 算法的改进。

设 x 和 y 是两个正整数,当 $\min(x)=\min(y)$,$\max(x)=\max(y)$且 $\text{sum}(x)=\text{sum}(y)$

时，称 x 和 y 是⊗等价的。由⊗运算的定义知，当 x 和 y 是⊗等价时，对于任意正整数 z 有 $(x⊗z)$ $(y⊗z)$ 且 $(z⊗x)$ $(z⊗y)$。由此可见，在 $S(N)$ 中只要关注⊗等价类中⊗运算次数最少的数。

⊗等价类可按 min，max，sum 进行分类。对于 $S(N)$ 中任一数 x 有

$0 \leqslant \min(x), \max(x) \leqslant 9; 1 \leqslant \text{sum}(x) \leqslant LL$，其中 $LL = 9\lceil \lg(L+1) \rceil$。

因此，可以用两个数组 leftn[1:LL] 和 rightn[0:9] 来存储 $S(N)$ 中⊗等价类，并由此构造出 $S(N)$ 中所有数。

输入 N 和 K 后，数组 leftn 和 rightn 由 init 初始化如下：

```
public static void init()
{
    biglen *=9;
    rightn=new int[10][10];
    for (int i=0;i<10;i++)
        for (int j=0;j<10;j++)
            rightn[i][j]=Integer.MAX_VALUE;
    leftn=new int[biglen+1];
    for (int i=1;i<=biglen;i++) leftn[i]=Integer.MAX_VALUE;
    a[0]=Integer.MAX_VALUE; a[1]=0; a[2]=0; leftn[0]=0;
    for(int i=0;i<len;i++)      count(ctoi(s1.charAt(i)),a);
    rightn[a[0]][a[1]]=0;sum=a[2];
    if (sum<=biglen) leftn[sum]=0;
}
```

上述程序中，count 和 trans 用来计算 $\min(i)$、$\max(i)$ 和 $\text{sum}(i)$。

```
public static void count(int i,int []a)
{
    if (i<a[0]) a[0]=i;
    if (i>a[1]) a[1]=i;
    a[2]+=i;
}

public static void trans(int t,int []a)
{
    a[0]=Integer.MAX_VALUE;a[1]=0;a[2]=0;
    while(t>0)
    {
        int j=t%10;
        t/=10;
        count(j,a);
    }
}
```

用⊗等价类思想实现的动态规划算法描述如下：

```
public static void compute()
```

```
{
    int best=Integer.MAX_VALUE;
    boolean flag=true;
    while(flag)
    {
        flag=false;
        for (int i=0;i<=biglen;i++)
            if (leftn[i]<Integer.MAX_VALUE)
                for (int j=0;j<10;j++)
                    for (int k=0;k<=j;k++)
                        if (rightn[k][j]<Integer.MAX_VALUE)
                        {
                            int num=i>0 ? i*j+k:sum*j+k;
                            trans(num,a);
                            int curr=leftn[i]+rightn[k][j]+1;
                            if (curr<leftn[a[2]]){leftn[a[2]]=curr;flag=true;}
                            if (curr<rightn[a[0]][a[1]])
                            {
                                rightn[a[0]][a[1]]=curr;flag=true;
                            }
                            if (num==kk && curr<best){best=curr;flag=true;}
                        }
    }
    if (best<Integer.MAX_VALUE) out(best);
    else out(-1);
}
```

完成整个计算的主函数 main 如下:

```
public static int kk,len,biglen,sum;
public static int []a=new int[3];
public static int []leftn;
public static int [][]rightn;
public static String s1,s2;
public static void main(String [] args)
{
    if(input()>0) init();
    else return;
    compute();
}
```

(9) 改进算法的计算复杂性。

改进算法所需空间显然为 $O(\log\log N)$。

改进算法所需计算时间为核心算法 compute 的计算时间。算法 compute 的 1 次 while 循环需要 $O(\log\log N)$ 的计算时间。最坏情况下需要 $O(\log N)$ 次 while 循环。因此,算法在最坏情况下所需的计算时间为 $O(\log N\log\log N)$。

新算法的计算复杂性有十分显著的改进。

算法实现题 3-12　最少费用购物

★ 问题描述

商店中每种商品都有标价。例如，一朵花的价格是 2 元。一个花瓶的价格是 5 元。为了吸引顾客，商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组，并降价销售。例如，3 朵花的价格不是 6 元而是 5 元。2 个花瓶加 1 朵花的优惠价是 10 元。试设计一算法，计算出某一顾客所购商品应付的最少费用。

★ 算法设计

对于给定欲购商品的价格和数量，以及优惠商品价，计算所购商品应付的最少费用。

★ 数据输入

由文件 input.txt 提供欲购商品数据。文件的第 1 行中有 1 个整数 $B(0\leqslant B\leqslant 5)$，表示所购商品种类数。接下来的 B 行，每行有 3 个数 C、K 和 P。C 表示商品的编码（每种商品有唯一编码），$1\leqslant C\leqslant 999$。$K$ 表示购买该种商品总数，$1\leqslant K\leqslant 5$。P 是该种商品的正常单价（每件商品的价格），$1\leqslant P\leqslant 999$。注意，一次最多可购买 $5\times 5=25$ 件商品。

由文件 offer.txt 提供优惠商品价数据。文件的第 1 行中有 1 个整数 $S(0\leqslant S\leqslant 99)$，表示共有 S 种优惠商品组合。接下来的 S 行，每行的第 1 个数描述优惠商品组合中商品的种类数 j。接着是 j 个数字对 (C,K)，其中 C 是商品编码，$1\leqslant C\leqslant 999$。$K$ 表示该种商品在此组合中的数量，$1\leqslant K\leqslant 5$。每行最后 1 个数字 $P(1\leqslant P\leqslant 9999)$ 表示此商品组合的优惠价。

★ 结果输出

将计算出的所购商品应付的最少费用输出到文件 output.txt 中。

输入文件示例		输出文件示例
input.txt	offer.txt	output.txt
2	2	14
7 3 2	1 7 3 5	
8 2 5	2 7 1 8 2 10	

分析与解答：

设 $\text{cost}(a,b,c,d,e)$ 表示购买商品组合 (a,b,c,d,e) 需要的最少费用。$A[k]$，$B[k]$，$C[k]$，$D[k]$，$E[k]$ 表示第 k 种优惠方案的商品组合。$\text{offer}(m)$ 是第 m 种优惠方案的价格。如果 $\text{cost}(a,b,c,d,e)$ 使用了第 m 种优惠方案，则：

$$\text{cost}(a,b,c,d,e)=\text{cost}(a-A[m],b-B[m],c-C[m],d-D[m],e-E[m])+\text{offer}(m)$$

即该问题具有最优子结构性质。按此递归式，容易设计解此问题的动态规划算法如下：

```
public static void minicost()
{
    int i,j,k,m,n,p,minm;
    minm=0;
    for(i=1;i<=b;i++)
        minm+=product[i]*purch[i].price;
    for(p=1;p<=s;p++)
    {
```

```
            i=product[1]-offer[p][1];
            j=product[2]-offer[p][2];
            k=product[3]-offer[p][3];
            m=product[4]-offer[p][4];
            n=product[5]-offer[p][5];
            if(i>=0 && j>=0 && k>=0 && m>=0 && n>=0 &&
                    (cost[i][j][k][m][n]+offer[p][0]<minm))
                minm=cost[i][j][k][m][n]+offer[p][0];
        }
        cost[product[1]][product[2]][product[3]][product[4]][product[5]]=minm;
    }
```

其中,product[i]是购买第 i 种商品的数量。由下面的 comp 迭代计算,init 作初始化计算。

```
public static void comp(int i)
{
    if(i>b){minicost();return;}
    for(int j=0;j<=purch[i].piece;j++){product[i]=j;comp(i+1);}
}

public static void init()
{
    int i,j,n,p,t,code;
    String fnm1="input5.txt";
    String fnm2="offer5.txt";
    ReadStreams fin=new ReadStreams(fnm1);
    ReadStreams fin1=new ReadStreams(fnm2);
    for(i=0;i<100;i++)
        for(j=0;j<6;j++) offer[i][j]=0;
    for(i=0;i<6;i++){purch[i]=new rec ();}
    for(i=0;i<6;i++){purch[i].piece=0;purch[i].price=0;product[i]=0;}
    b=fin.readInt();
    for(i=1;i<=b;i++)
    {
        code=fin.readInt();
        purch[i].piece=fin.readInt();
        purch[i].price=fin.readInt();
        num[code]=i;
    }
    s=fin1.readInt();
    for(i=1;i<=s;i++)
    {
        t=fin1.readInt();
        for(j=1;j<=t;j++)
        {
            n=fin1.readInt();
            p=fin1.readInt();
            offer[i][num[n]]=p;
```

```
            }
            offer[i][0]=fin1.readInt();
        }
    }
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    init();
    comp(1);
    out();
}
```

算法实现题 3-13　最大长方体问题

★ 问题描述

一个长、宽、高分别为 m, n, p 的长方体被分割成 $m \times n \times p$ 个小立方体。每个小立方体内有一个整数。试设计一个算法，计算出所给长方体的最大子长方体。子长方体的大小由它所含所有整数之和确定。

★ 算法设计

对于给定的长、宽、高分别为 m, n, p 的长方体，计算最大子长方体的大小。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是 3 个正整数 m, n, p，$1 \leqslant m, n, p \leqslant 50$。接下来的 $m \times n$ 行每行 p 个正整数，表示小立方体中的数。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的最大子长方体的大小。

输入文件示例	输出文件示例
input.txt	output.txt
3　3　3	14
0　−1　2	
1　2　2	
1　1　−2	
−2　−1　−1	
−3　3　−2	
−2　−3　1	
−2　3　3	
0　1　3	
2　1　−3	

分析与解答：

在最大子矩阵和问题的动态规划算法基础上，容易设计解此问题的动态规划算法如下：

```
public static int maxSum3(int [][][]a,int m,int n,int p)
{
    int max,sum=0;
    int [][]b=new int[n][p];
    for (int i=0;i<m;i++)
    {
        for (int k=0;k<n;k++)
            for(int t=0;t<p;t++) b[k][t]=0;
        for (int j=i;j<m;j++)
        {
            for (int k=0;k<n;k++)
                for(int t=0;t<p;t++)
                    b[k][t]+=a[j][k][t];
            max=maxSum2(b,n,p);
            if (max>sum)sum=max;
        }
    }
    return sum;
}
```

算法实现题 3-14　正则表达式匹配问题

★ 问题描述

许多操作系统都采用正则表达式实现文件匹配功能。一种简单的正则表达式由英文字母、数字及通配符*和?组成。?代表任意一个字符,*则可以代表任意多个字符。现要用正则表达式对部分文件进行操作。

试设计一个算法,找出一个正则表达式,使其能匹配的待操作文件最多,但不能匹配任何不进行操作的文件。所找出的正则表达式的长度还应是最短的。

★ 算法设计

对于给定的待操作文件,找出一个能匹配最多待操作文件的正则表达式。

★ 数据输入

由文件 input.txt 提供输入数据。文件由 $n(1\leq n\leq 250)$ 行组成。每行给出一个文件名。文件名由英文字母和数字组成。英文字符要区分大小写,文件名长度不超过8个字符。文件名后是一个空格符和一个字符+或-。+表示要对该行给出的文件进行操作,-表示不进行操作。

★ 结果输出

将计算出的最多文件匹配数和最优正则表达式输出到文件 output.txt 中。文件的第1行中的数是计算出的最多文件匹配数。文件的第1行是最优正则表达式。

输入文件示例	输出文件示例
input.txt	output.txt
EXCHANGE +	3
EXTRA +	*A*

HARDWARE +
MOUSE −
NETWORK −

分析与解答：

设当前考查的正则表达式为 s，当前期考查的文件为 f。用 match(i,j)表示 $s[1..i]$与 $f[1..j]$的匹配情况。当 $s[1..i]$能匹配 $f[1..j]$时 match$(i,j)=1$，否则 match$(i,j)=0$。显然，可用下面的递归式计算 match(i,j)。

$$\text{match}(i,j)=\begin{cases}1 & \begin{cases}\text{match}(i-1,j-1)=1, s[i]='?'\\ \text{match}(i-1,j-1)=1, s[i]=f[j]\\ \text{match}(i-1,k)=1, s[i]='*'\end{cases}\\ 0\end{cases}$$

据此可设计求最优匹配的回溯法如下：

```
public static void search(int len)
{
    if((currmat>maxmat || currmat==maxmat&&len<minlen)&&ok(len))
    {
        maxmat=currmat;minlen=len;
        for(int i=0;i<=MAXL;i++)minmat[i]=s[i];
    }
    len++;
    if(len==1 || s[len-1]!='*')
    {
        s[len]='?';
        if(check(len)) search(len);
        s[len]='*';
        if(check(len)) search(len);
    }
    for(int i=1;i<=p[len-1];i++)
    {
        s[len]=cha[len-1][i].c;
        if(check(len)) search(len);
    }
}
```

上述程序中，check 用于计算当前匹配情况，ok 用于判定是否匹配非操作文件。

```
public static boolean check(int len)
{
    int i,j,t,k=0;
    currmat=0;
    for(i=1;i<=n[0];i++)
    {
        for(j=0;j<=MAXFL;j++)match[len][i][j]=0;
```

```
            if(len==1 && s[1]=='*') match[len][i][0]=1;
            for(j=1;j<=f[i].length();j++)
                switch(s[len])
                {
                    case '*':
                        for(t=0;t<=j;t++)
                            if(match[len-1][i][t]==1) {match[len][i][j]=1;break;}
                        break;
                    case '?':
                        match[len][i][j]=match[len-1][i][j-1];
                        break;
                    default:
                        if(s[len]==f[i].charAt(j-1))
                            match[len][i][j]=match[len-1][i][j-1];
                        break;
                }
            for(j=f[i].length();j>=1;j--)
                if(match[len][i][j]==1){
                    k++;
                    if(j==f[i].length())currmat++;
                    break;
                }
        }
        if (k<maxmat || k==maxmat && len>=minlen) return false;
        p[len]=0;
        for(i=1;i<=n[0];i++)
            for(j=1;j<=f[i].length()-1;j++)
                if (match[len][i][j]==1) save(f[i].charAt(j),len);
        return true;
    }

    public static boolean ok(int len)
    {
        int i,j,k,t;
        for(k=1;k<=len;k++)
            for(i=n[0]+1;i<=n[0]+n[1];i++)
            {
                for(j=0;j<=MAXFL;j++)match[k][i][j]=0;
                if(s[1]=='*' && k==1) match[k][i][0]=1;
                for(j=1;j<=f[i].length();j++)
                    switch(s[k])
                    {
                        case '*':
                            for (t=0;t<=j;t++)
                                if (match[k-1][i][t]==1){match[k][i][j]=1;break;}
                            break;
```

```
                    case '?':
                        match[k][i][j]=match[k-1][i][j-1];
                        break;
                    default:
                    if (s[k]==f[i].charAt(j-1))match[k][i][j]=match[k-1][i][j-1];
                    break;
                }
            }
    for(i=n[0]+1;i<=n[0]+n[1];i++)
        if(match[len][i][f[i].length()]==1)return false;
    return true;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
    readin();
    search(0);
    out();
}
```

readin 读入数据并初始化。

```
public static void readin()
{
    String []k=new String[MAXN+1];
    String str,chr;
    ReadStream keyboard=new ReadStream();
    n[0]=0;n[1]=0;p[0]=0;
    for(int i=0;i<=MAXL;i++)
        for(int j=0;j<=MAXP;j++) cha[i][j]=new reco();
    while(true)
    {
        str=keyboard.readString();
        if(str.length()==0)break;
        chr=keyboard.readString();
        if (chr.charAt (0)=='+'){f[++n[0]]=str;save(str.charAt (0),0);}
        else k[++n[1]]=str;
    }
    for(int i=1;i<=n[1];i++) f[n[0]+i]=k[i];
    for(int i=0;i<=MAXL;i++)
        for(int j=0;j<=MAXN;j++)
            for(int l=0;l<=MAXFL;l++)    match[i][j][l]=0;
    for(int i=1;i<=n[0]+n[1];i++) match[0][i][0]=1;
    maxmat=0;minlen=255;
```

程序中 save 对操作文件名中出现的字符按出现频率排序存储，以加快搜索进程。

```
public static void save(char c,int len)
{
    for (int i=1;i<=p[len];i++)
        if(cha[len][i].c==c)
        {
            cha[len][i].f++;
            cha[len][0]=cha[len][i];
            int j=i;
            while(cha[len][j-1].f<cha[len][0].f)
            {
                cha[len][j]=cha[len][j-1];j--;
            }
            cha[len][j]=cha[len][0];
            return;
        }
    cha[len][++p[len]].c=c;cha[len][p[len]].f=1;
}
```

算法实现题 3-15　双调旅行售货员问题

★ 问题描述

欧几里得旅行售货员问题是对给定的平面上 n 个点确定一条连接这 n 个点的长度最短的哈密顿回路。由于欧几里得距离满足三角不等式，所以欧几里得旅行售货员问题是一个特殊的具有三角不等式性质的旅行售货员问题。它仍是一个 NP 完全问题。最短双调 TSP 回路是欧几里得旅行售货员问题的特殊情况。平面上 n 个点的双调 TSP 回路是从最左点开始，严格地由左至右直到最右点，然后严格地由右至左直至最左点，且连接每一个点恰好一次的一条闭合回路。

★ 算法设计

给定平面上 n 个点，计算这 n 个点的最短双调 TSP 回路。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有一个正整数 n，表示给定的平面上的点数。接下来的 n 行中，每行两个实数，分别表示点的 x 坐标和 y 坐标。

★ 结果输出

将计算的最短双调 TSP 回路的长度(保留 2 位小数)输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7	25.58
0 6	
1 0	
2 3	
5 4	

6 1
7 5
8 2

分析与解答：

首先将给定的平面上 n 个点依其 x 坐标排序。设排好序的 n 个点为 $p_i=(x_i,y_i)$，$i=1,2,\cdots,n$。点集 $\{p_1,p_2,\cdots,p_i\}$ 的最短双调 TSP 回路的长度记作 $t(i)$。容易证明该问题具有最优子结构性质。$t(i)$ 满足如下动态规划递归式

$$t(i) = \min_{1<k<i} \{t(k) + D(k,i) + d(k-1,i) - d(k-1,k)\}$$

$$t(1) = 0, \quad t(2) = 2d(1,2)$$

其中，$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$，$D(i,j) = \sum_{k=i+1}^{j} d(k-1,k)$。

设 $s(i) = \sum_{k=2}^{i} d(k-1,k)$，则 $D(k,i) = s(i) - s(k)$，$d(k-1,k) = s(k) - s(k-1)$。

上述递归式可进一步简化为

$$t(i) = \min_{1<k<i} \{t(k) + s(i) + s(k-1) - 2s(k) + d(k-1,i)\}$$

按此递归式计算出的 $t(n)$ 为最优值。算法所需的计算时间为 $O(n^2)$。

```
public static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    point=new Element[n];
    for (int i=0; i < n; i++)
    {
        double x=keyboard.readDouble();
        double y=keyboard.readDouble();
     point[i]=new Element (x,y);
    }
    MergeSort.mergeSort (point);
}

public static void dynamic()
{
    s[1]=0;
    for (int i=2; i<=n; i++) s[i]=dist(i-1,i)+s[i-1];
    t[2]=2 * s[2];
    for(int i=3;i<=n;i++)
    {
        t[i]=t[2]+s[i]-2 * s[2]+dist(i,1);
        for (int j=2;j<=i-2;j++)
        {
            double temp=t[j+1]+s[i]+s[j]-2 * s[j+1]+dist(i,j);
            if(t[i]>temp)t[i]=temp;
        }
```

```
        }
    }
```

程序中的 dist 计算两点之间的距离。

```
    public static double dist(int p,int q)
    {
        return Math.sqrt(Math.pow(point[p-1].x-point[q-1].x,2)+
                         Math.pow(point[p-1].y-point[q-1].y,2));
    }
```

算法实现题 3-16　最大 k 乘积问题

★ 问题描述

设 I 是一个 n 位十进制整数。如果将 I 划分为 k 段，则可得到 k 个整数。这 k 个整数的乘积称为 I 的一个 k 乘积。试设计一个算法，对于给定的 I 和 k，求出 I 的最大 k 乘积。

★ 算法设计

对于给定的 I 和 k，计算 I 的最大 k 乘积。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中有两个正整数 n 和 k。其中，正整数 n 是序列的长度，正整数 k 是分割的段数。第 2 行中是一个 n 位十进制整数，$n \leqslant 10$。

★ 结果输出

将计算结果输出到文件 output.txt 中。文件的第 1 行中的数是计算出的最大 k 乘积。

输入文件示例	输出文件示例
input.txt	output.txt
2 1	15
15	

分析与解答：

设 $I(s,t)$ 是 I 的从 s 位开始的 t 位的数字组成的十进制数。$f(i,j)$ 表示 $I(0,i)$ 的最大 j 乘积，则 $f(i,j)$ 具有最优子结构性质。计算 $f(i,j)$ 的动态规划递归式如下：

$$f(i,j) = \max_{1 \leqslant k < i} \{f(k,j-1) * I(k,i-k)\}$$

据此可设计最大 k 乘积问题的动态规划算法如下：

```
    public static void solve(int n,int m)
    {
        int i,j,k;
        int temp,maxt,tk=0;
        for(i=1; i<=n; i++) f[i][1]=conv(0,i);
        for(j=2;j<=m;j++)
            for(i=j;i<=n;i++)
            {
                for(k=1,temp=0;k<i;k++)
```

```
                {
                    maxt=f[k][j 1]*conv(k,i k);
                    if(temp<maxt){temp=maxt;tk=k;}
                }
                f[i][j]=temp;ka[i][j]=tk;
            }
}
```

程序中的 conv 计算 $I(s,t)$的值。

```
public static int conv(int i,int j)
{
    String str1=str.substring(i,i+j);
    return Integer.parseInt (str1);
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int m=keyboard.readInt();
    if ((n<m) || (n==0)) {System.out.println(0);return;}
    f=new int[n+1][m+1];
    ka=new int[n+1][m+1];
    str=keyboard.readString();
    if (n!=str.length()) {System.out.println(0);return;}
    solve(n,m);
    out(n,m);
}
```

out 输出计算结果。

```
public static void out(int n,int m)
{
    System.out.println(f[n][m]);
    for(int i=m,k=n;i>=1 && k>0;k=ka[k][i],i--)
        System.out.println("f["+k+"]["+i+"]="+f[k][i]);
```

第 4 章 贪心算法

习题 4-1　活动安排问题的贪心选择

在活动安排问题中，还可以有其他的贪心选择方案，但并不能保证产生最优解。给出一个例子，说明选择具有最短时段的相容活动作为贪心选择，得不到最优解。选择覆盖未选择活动最少的相容活动作为贪心选择，也得不到最优解。

分析与解答：

(1) 在图 4-1 给出的 11 个活动中，若选择具有最短时段的相容活动作为贪心选择，则得到的解为{1,4,5}。

(2) 若选择覆盖未选择活动最少的相容活动作为贪心选择，则首先选择活动 5。而一旦选择了活动 5，余下的活动中最多还有两个相容活动可选择。因此，按这个策略最多只能安排 3 个活动。

(3) 最优解显然为{1,2,3,4}，故(1)和(2)中的解均不是最优解。

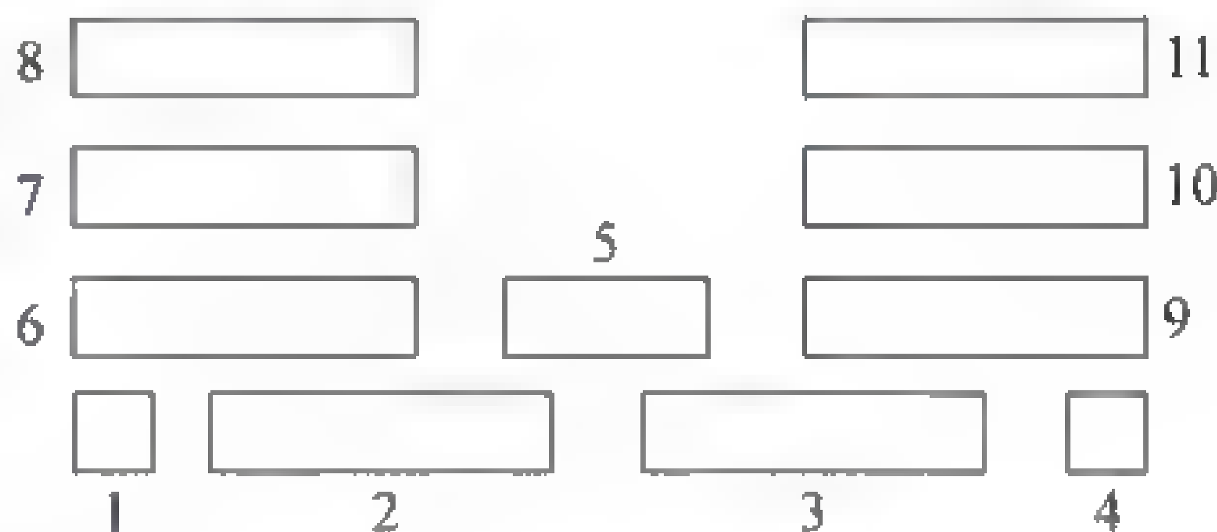


图 4-1　活动安排示例

习题 4-2　背包问题的贪心选择性质

证明背包问题具有贪心选择性质。

分析与解答：

背包问题可描述为：给定 $W>0$，$w_i>0$，$v_i>0$，$1\leqslant i\leqslant n$，要求找出一个 n 元向量 $(x_1,x_2,\cdots,x_n)$，$0\leqslant x_i\leqslant 1$，$1\leqslant i\leqslant n$，使得 $\sum_{i=1}^{n} w_i x_i \leqslant W$，而且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。

对于教材中解该问题的贪心算法所采用的贪心选择策略，其贪心选择性质可描述为：

若 $\frac{v_i}{w_i}\geqslant\frac{v_{i+1}}{w_{i+1}}$，$1\leqslant i\leqslant n-1$，则存在背包问题的一个最优解 $(x_1,x_2,\cdots,x_n)$，使得 $x_1=\min\left\{\frac{W}{w_1},1\right\}$。

下面分 3 种情形证明背包问题的这个贪心选择性质。

(1) $\sum_{i=1}^{n} w_i\leqslant W$ 的情形。此时唯一的最优解为$(x_1,x_2,\cdots,x_n)$。

(2) $\sum_{i=1}^{n} w_i > W$，且 $\frac{v_1}{w_1} = \frac{v_i}{w_i}$，$2 \leqslant i \leqslant n$ 的情形。

(3) $\sum_{i=1}^{n} w_i > W$ 的情形。

习题 4-3　特殊的 0-1 背包问题

若在 0-1 背包问题中，各物品依重量递增排列时，其价值恰好依递减序排列。对这个特殊的 0-1 背包问题，设计一个有效算法找出最优解，并说明算法的正确性。

分析与解答：

设所给的输入为 $W>0$，$w_i>0$，$v_i>0$，$1 \leqslant i \leqslant n$。不妨设 $0 < w_1 \leqslant w_2 \leqslant \cdots \leqslant w_n$。由题意知 $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_n > 0$。由此可知 $\frac{v_i}{w_i} \geqslant \frac{v_{i+1}}{w_{i+1}}$，$1 \leqslant i \leqslant n-1$。

贪心选择性质：

当 $w_1 > W$ 时问题无解。

当 $w_1 \leqslant W$ 时，存在 0-1 背包问题的一个最优解 $S \subseteq \{1,2,\cdots,n\}$ 使得 $1 \in S$。

事实上，设 $S \subseteq \{1,2,\cdots,n\}$ 是 0-1 背包问题的一个最优解，且 $1 \notin S$。对任意 $i \in S$，取 $S_i = S \cup \{1\} - \{i\}$，则 S_i 满足贪心选择性质的最优解。

习题 4-4　程序最优存储问题

假定要把长为 $l_1, l_2, \cdots, l_n$ 的 n 个程序放在磁带 T_1 和 T_2 中，并且希望按照使最大检索时间取最小值的方式存放，即如果存放在 T_1 和 T_2 上的程序集合分别是 A 和 B，则希望所选择的 A 和 B 使得 $\max\left\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\right\}$ 取最小值。贪心算法：开始将 A 和 B 都初始化为空，然后一次考虑一个程序，如果 $\sum_{i \in A} l_i = \min\left\{\sum_{i \in A} l_i, \sum_{i \in B} l_i\right\}$，则将当前正在考虑的那个程序分配给 A，否则分配给 B。证明无论是按 $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n$ 还是按 $l_1 \geqslant l_2 \geqslant \cdots \geqslant l_n$ 的次序来考虑程序，这种方法都不能产生最优解。应当采用什么策略？写出一个完整的算法并证明其正确性。

分析与解答：

设变量 $x_i = 1$ 表示将 l_i 存放在 T_1 上，且 T_1 的检索时间较短，则

$$\sum_{i=1}^{n} l_i x_i - \sum_{i=1}^{n} l_i(1-x_i) \leqslant 0, \quad 2\sum_{i=1}^{n} l_i x_i \leqslant \sum_{i=1}^{n} l_i, \quad \sum_{i=1}^{n} l_i x_i \leqslant \frac{1}{2}\sum_{i=1}^{n} l_i$$

T_1 的检索时间应取最大值，因此问题归结为

$$\max \sum_{i=1}^{n} l_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} l_i x_i \leqslant \frac{1}{2}\sum_{i=1}^{n} l_i$$

这与第 5 章中的装载问题等价，是一个特殊的 0-1 背包问题。

习题 4-5　最优装载问题的贪心算法

将最优装载问题的贪心算法推广到两艘船的情形，贪心算法仍能产生最优解吗？

分析与解答：

贪心算法不能产生最优解，见第 5 章的装载问题。

习题 4-6　Fibonacci 序列的 Huffman 编码

字符 a～h 出现的频率恰好是前 8 个 Fibonacci 数,它们的 Huffman 编码是什么? 将结果推广到 n 个字符的频率恰好是前 n 个 Fibonacci 数的情形。

分析与解答:

频率恰好是前 8 个 Fibonacci 数的哈夫曼编码树如图 4-2 所示。

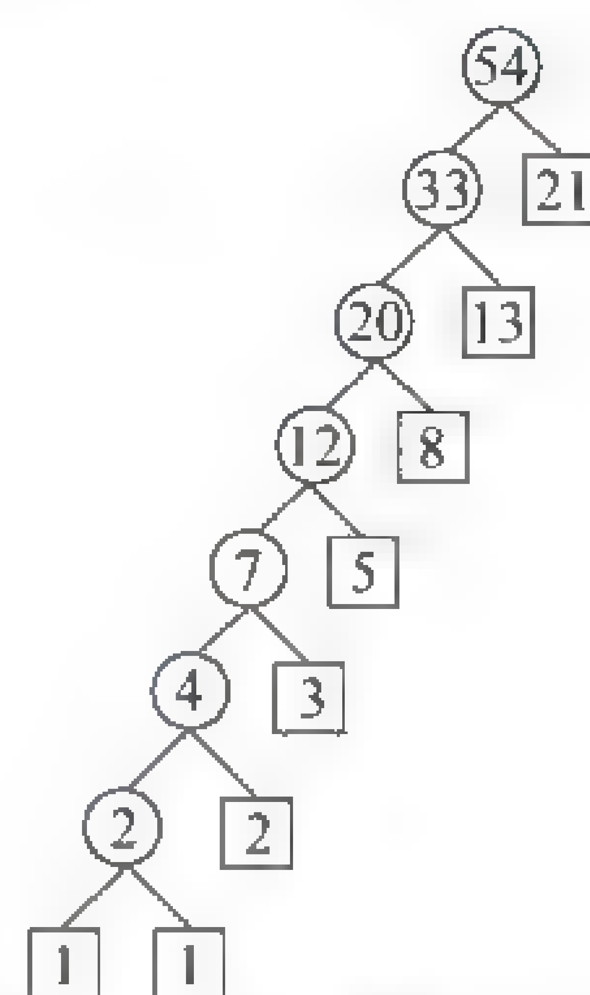


图 4-2　哈夫曼编码树

在一般情况下,n 个字符的频率恰好是前 n 个 Fibonacci 数,则相应的哈夫曼编码树深度为 $n-1$,第一个字符的编码长度为 $n-1$。自底向上第 i 个圆结点中的数为 $\sum_{k=0}^{i} f_k$。用数学归纳法容易证明 $\sum_{k=0}^{i} f_k < f_{i+2}$。该性质保证了频率恰好是前 n 个 Fibonacci 数的哈夫曼编码树具有所述形状和性质。

习题 4-7　最优前缀码的编码序列

设 $C=\{0,1,\cdots,n-1\}$ 是 n 个字符的集合。证明关于 C 的任何最优前缀码可以表示为长度为 $2n-1+n\lceil \log n\rceil$ 位的编码序列。(提示:用 $2n-1$ 位描述树结构)

分析与解答:

任何最优前缀码所相应的编码二叉树是一棵完全二叉树,有 n 个叶结点和 $n-1$ 个内结点。用 1 位表示 1 个结点的类型,1 表示内结点,0 表示叶结点,总共需 $2n-1$ 位。对编码树的前序遍历可以唯一表示该编码树结构。

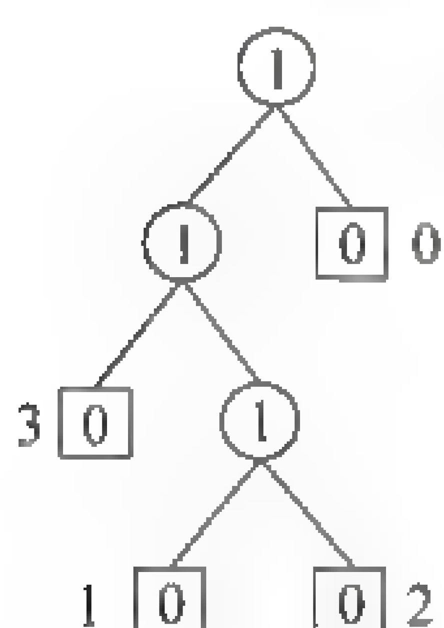


图 4-3　编码树结构

例如,当 $n=4$ 时,图 4-3 所示编码树结构可以唯一表示为 1101000。

在每个叶结点后,即每个 0 后面紧跟 $\lceil \log n\rceil$ 位表示该叶结点处的数字,即可完整表示整棵编码树。例如,图 4-3 中的编码树可表示为 110111001010000。由此可知,在一般情况下,最优前缀码可以表示为长度为 $2n-1+n\lceil \log n\rceil$ 位的编码序列。

习题 4-8　任务集独立性问题

说明如何用引理 4.6 的性质(2),在 $O(|A|)$ 时间里确定给定的任务集 A 是否独立。

分析与解答:

由引理 4.6 的性质(2)可知,对任意 $A[1:k]\subseteq\{1,2,\cdots,n\}$,当 $N_t(A)\leqslant t, t=1,2,\cdots,n$ 时,任务子集 A 是独立的。用桶排序算法可在 $O(|A|)$ 时间内完成这一判定。

习题 4-9　矩阵拟阵

给定 $n\times n$ 实值矩阵 $\boldsymbol{T}$,证明 (S,I) 是拟阵。其中,S 是 $\boldsymbol{T}$ 的列向量的集合,$A\in I$ 当且仅当 A 中的列是线性独立的。

分析与解答:

由线性空间理论中的基交换定理容易证明 I 满足拟阵的交换性质(3),从而证明 (S,I) 是拟阵。

习题 4-10 最小权最大独立子集拟阵

说明如何变换带权拟阵的权函数，使最小权最大独立子集问题变换为等价的标准带权拟阵问题，并证明变换的正确性。

分析与解答：

设带权拟阵 $M=(S,I)$ 的权函数为 W。令 $W_0=\max\{W(x)\mid x\in S\}+1$。定义 $M=(S,I)$ 的另一权函数为 $W'(x)=W_0-W(x)$。则容易证明权函数为 W 的最小权最大独立子集问题等价于权函数为 W' 的标准带权拟阵问题。

习题 4-11 整数边权 Prim 算法

假设具有 n 个顶点的连通带权图中所有边的权值均为 $1\sim n$ 的整数，能对 Kruskal 算法作何改进，时间复杂性能改进到何种程度？若对某常量 N，所有边的权值均为 $1\sim N$ 的整数，在这种情况下又如何？在上述两种情况下，对 Prim 算法能作何改进？

分析与解答：

Prim 算法的主要运算在于寻找顶点分别在 $V-U$ 与 U 中边权最小的边，找出这样的边后再对 U，lowcost 和 closest 进行调整。如果用一个优先队列来完成这些工作，则要求优先队列支持 DeleteMin 和 Decrease-Key 运算。如果采用 Fibonacci 堆来实现优先队列，则可以在 $O(|V|\log|V|)$时间内完成对优先队列的所有操作。因此 Prim 算法可在 $O(|E|+|V|\log|V|)$时间内完成。如果边权是 $1\sim N$（常数）的整数，则可用一个数组 $Q[0:N+1]$ 来表示优先队列。其中 $N+1$ 单元存储 $+\infty$，其余各单元存储顶点的双链表，其权值等于数组下标。按照此策略，DeleteMin 和 Decrease-Key 都只要 $O(1)$ 时间，从而可在 $O(|E|)$时间内完成 Prim 算法。

如果边权是 $1\sim n$ 的整数，则上述方法就不适用了。此时，Decrease-Key 仍可在 $O(1)$ 时间内完成，但 DeleteMin 运算需要 $O(n)$ 时间。完成 Prim 算法就需要 $O(|E|+n^2)$时间。如果用 van Emde Boas 优先队列，则可在 $O(n\log\log n)$ 时间内完成所有优先队列操作，从而可在 $O(|E|+n\log\log n)$时间内完成 Prim 算法。

习题 4-12 最大权最小生成树

试设计一个构造图 G 生成树的算法，使得构造出的生成树的边的最大权值达到最小。

分析与解答：

对于这个问题，关键的一点是注意到任何一棵最小生成树都使边的最大权值达到最小。事实上，设 T 是 G 的一棵最小生成树，T' 是 G 的一棵使最大权值达到最小的生成树。e 是 T 中的最大权边，e' 是 T' 中的最大权边，且 $w(e')<w(e)$。将 e 从 T 中删去后，T 将分为两个连通分支。此时，一定有 T' 中的边 e'' 连接这两个连通分支，否则 T' 将是不连通的。将 e'' 加入 T 的这两个连通分支将得到一棵新的生成树 $T''=T-e+e''$。由于 e' 是 T' 的最大权边，故 $w(e'')\leqslant w(e')<w(e)$，从而有 $w(T'')=w(T)-w(e)+w(e'')<w(T)$。这与 T 是最小生成树相矛盾。

通过以上的讨论可知，用 Prim 算法或 Kruskal 算法均可构造出 G 的最大权值达到最小的生成树。

习题 4-13 最短路径的负边权

试举例说明如果允许带权有向图中某些边的权为负实数，则 Dijkstra 算法不能正确求

得从源到所有其他顶点的最短路径长度。

分析与解答：

对于图4-4所示的有向图G，用Dijkstra算法找顶点1到顶点3的最短路径为1,3，其长度为1，而实际上最短路径应为1,2,3，其长度为0。可见，当有向图G中含有负权边时，Dijkstra算法不能正确工作。

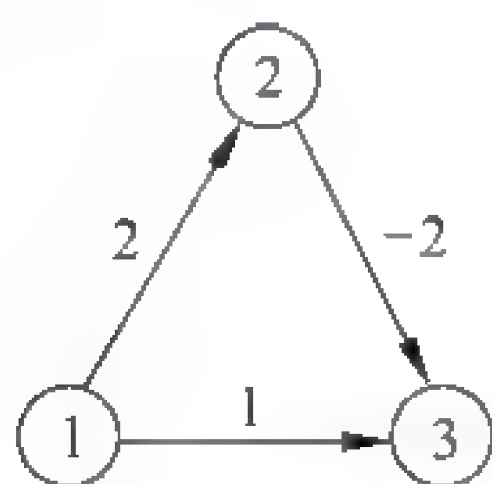


图4-4　负边权有向图

习题4-14　整数边权Dijkstra算法

设G是具有n个顶点和e条边的带权有向图，各边的权值为$0\sim N-1$的整数，N为非负整数。修改Dijkstra算法使其能在$O(Nn+e)$时间内计算出从源到所有其他顶点之间的最短路径长度。

分析与解答：

Dijkstra算法的关键在于选取$V-S$中顶点w，使$D[w]=\min\{D[v]|v\in V-S\}$。当各边的权值是$0\sim N-1$的整数时，首先注意到，对任意$u\in S,v\in V-S$有$D[u]\leqslant D[v]$。其次，对任意$u,v\in V-S$，且$D[u]<\infty,D[v]<\infty$时，有$|D[u]-D[v]|\leqslant N-1$。

事实上，设$u\in V-S$，且$D[u]=\min\{D[w]|w\in V-S\}$，$v\in V-S$是任意的。由于$D[v]<\infty$，故存在$w\in S$，使$D[v]\leqslant D[w]+C(w,v)\leqslant D[u]+N-1$。由此即知，对任意的$u,v\in V-S$有$|D[u]-D[v]|\leqslant N-1$。换句话说，在任一时刻，$V-S$中顶点最多有$N$个不同的$D$值。即在任一时刻，总存在常数$a$，使得对任意$v\in V-S$，有$a\leqslant D[v]\leqslant a+N-1$。由此想到可用桶排序算法的思想，设置$N$个桶来存放$V-S$中的元素，使得当$v\in V-S$时，将顶点$v$存放在$D[v]$号桶中。这里可能遇到的一个问题是，在算法执行过程中，a值可能不断增大。为了解决这个问题，可以设置一个游标k指示出当前最小桶编号的位置。$V-S$中元素最多存放在其后的$N-1$个桶中。当顺序的N个桶超出桶头数组的界时，可用循环数组的思想再从头开始存放。

对整数边权有向图，算法除了对链表的处理时间外，需要$O(N)$计算时间。算法对每条边处理一次，故算法中对链表处理所需的总时间为$O(e)$。因此，对边权值在$0\sim N-1$的有向图，算法需要$O(Nn+e)$计算时间。

算法实现题4-1　会场安排问题

★ 问题描述

假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的贪心算法进行安排(这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点，则不相容活动间可用边相连。使相邻顶点着有不同颜色的最小着色数，相应于找最小会场数)。

★ 算法设计

对于给定的k个待安排的活动，计算使用最少会场的时间表。

★ 数据输入

由文件input.txt给出输入数据。第1行有1个正整数k，表示有k个待安排的活动。接下来的k行中，每行有2个正整数，分别表示k个待安排的活动开始时间和结束时间。时间以0点开始的分钟计。

★ 结果输出

将计算出的最少会场数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	3
1 23	
12 28	
25 35	
27 80	
36 50	

分析与解答：

设所给的 n 个活动为 $1,2,\cdots,n$，它们的开始时间和结束时间分别为 $s[i]$ 和 $f[i]$，$i=1\sim n$，且 $f[1]<f[2]<\cdots<f[n]$。

(1) 容易想到用算法 GreedySelector 来安排会场。在最坏情况下算法需要 $O(n^2)$ 计算时间。

(2) 实际上，可以设计出一个更有效的算法。将 n 个活动 $1,2,\cdots,n$ 看作实直线上的 n 个半闭活动区间$[s[i],f[i])$，$i=1\sim n$。所讨论的问题实际上是求这 n 个半闭区间的最大重叠数。重叠的活动区间所相应的活动是互不相容的。若这 n 个活动区间的最大重叠数为 m，则这 m 个重叠区间所对应的活动互不相容，因此至少要安排 m 个会场来容纳这 m 个活动。

为了有效地对这 n 个活动进行安排，首先将 n 个活动区间的 $2n$ 个端点排序，然后用扫描算法从左到右扫描整个实直线。在每个事件点处(即活动的开始时刻或结束时刻)作会场安排。当遇到一个开始时刻 $s[i]$时，将活动 i 安排在一个空闲的会场中；当遇到一个结束时刻 $f[i]$时，将活动 i 占用的会场释放到空闲会场栈中，以备使用。经过这样一遍扫描，最多安排 m 个会场(m 是最大重叠区间数)。因此所作的会场安排是最优的。上述算法所需的计算时间主要用于对 $2n$ 个区间端点的排序，这需要 $O(n\log n)$ 计算时间。

具体算法描述如下：

```
public static int greedy(point []x)
{
    int sum=0,curr=0,n=x.length;
    MergeSort.mergeSort (x);
    for(int i=0;i<n;i++)
    {
        if (x[i].leftend())curr++;
        else curr--;
        //处理 x[i]=x[i+1]的情况
        if((i==n-1 || x[i].compareTo(x[i+1]) < 0) && curr>sum)sum=curr;
    }
    return sum;
}
```

算法实现题 4-2 最优合并问题

★ 问题描述

给定 k 个排好序的序列 $s_1, s_2, \cdots, s_k$，用 2 路合并算法将这 k 个序列合并成一个序列。假设所采用的 2 路合并算法合并 2 个长度分别为 m 和 n 的序列需要 $m+n-1$ 次比较。试设计一个算法确定合并这个序列的最优合并顺序，使所需的总比较次数最少。

为了进行比较，还需要确定合并这个序列的最差合并顺序，使所需的总比较次数最多。

★ 算法设计

对于给定的 k 个待合并序列，计算最多比较次数和最少比较次数合并方案。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k，表示有 k 个待合并序列。第 2 行中有 k 个正整数，表示 k 个待合并序列的长度。

★ 结果输出

将计算出的最多比较次数和最少比较次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4	78 52
5 12 11 2	

分析与解答：

见教材中的 Huffman 算法。

算法实现题 4-3 磁带最优存储问题

★ 问题描述

设有 n 个程序 $\{1,2,\cdots,n\}$ 要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 l_i，$1 \leqslant i \leqslant n$。这 n 个程序的读取概率分别是 $p_1, p_2, \cdots, p_n$，且 $\sum_{i=1}^{n} p_i = 1$。如果将这 n 个程序按 $i_1, i_2, \cdots, i_n$ 的次序存放，则读取程序 i_r 所需的时间 $t_r = c\sum_{k=1}^{r} p_{i_k} l_{i_k}$。这 n 个程序的平均读取时间为 $\sum_{r=1}^{n} t_r$。

磁带最优存储问题要求确定这 n 个程序在磁带上的一个存储次序，使平均读取时间达到最小。试设计一个解此问题的算法，并分析算法的正确性和计算复杂性。

★ 算法设计

对于给定的 n 个程序存放在磁带上的长度和读取概率，计算 n 个程序的最优存储方案。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示文件个数。接下来的 n 行中，每行有两个正整数 a 和 b，分别表示程序存放在磁带上的长度和读取概率。实际上第 k 个程序的读取概率 $a_k / \sum_{i=1}^{n} a_i$。对所有输入均假定 $c=1$。

★ 结果输出

将计算出的最小平均读取时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	85.6193
71 872	
46 452	
9 265	
73 120	
35 85	

分析与解答：

贪心策略：最短平均读取时间程序优先。

```
public static double greedy(int []x,int []p)
{
    int n=p.length;
    int []y=new int[n];
    for(int i=0;i<n;i++) y[i]=x[i] * p[i];
    QuickSort.quickSort (y,n);
    for(int i=1;i<n;i++)y[i]+=y[i-1];
    double m=0,t=0;
    for(int i=0;i<n;i++){m+=p[i];t+=y[i];}
    return t/m;
}
```

算法实现题 4-4 磁盘文件最优存储问题

★ 问题描述

设磁盘上有 n 个文件 $f_1,f_2,\cdots,f_n$，每个文件占磁盘上 1 个磁道。这 n 个文件的检索概率分别是 $p_1,p_2,\cdots,p_n$，且 $\sum_{i=1}^{n} p_i=1$。磁头从当前磁道移到被检信息磁道所需的时间可用这 2 个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i 道上，$1\leqslant i\leqslant n$，则检索这 n 个文件的期望时间是 $\sum_{1\leqslant i<j\leqslant n} p_i p_j d(i,j)$。其中，$d(i,j)$ 是第 i 道与第 j 道之间的径向距离 $|i-j|$。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置，使期望检索时间达到最小。试设计一个解此问题的算法，并分析算法的正确性与计算复杂性。

★ 算法设计

对于给定的文件检索概率，计算磁盘文件的最优存储方案。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示文件个数。第 2 行有 n 个正整数 a_i，表示文件的检索概率。实际上第 k 个文件的检索概率应为 $a_k\Big/\sum_{i=1}^{n} a_i$。

★ 结果输出

将计算出的最小期望检索时间输出到文件 output. txt。

输入文件示例	输出文件示例
input. txt	output. txt
5	0.547396
33 55 22 11 9	

分析与解答:

将 n 个文件按其概率排序。设排序后有 $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$。

贪心策略:f_1 占中心磁道,f_2 和 f_3 分居 f_1 的两侧,f_4 在 f_2 的左侧,f_5 在 f_3 的右侧,……。

具体算法实现如下:

```
public static double greedy(int []p)
{
    int n=p.length;
    int []x=new int[n];
    QuickSort.quickSort(p,n);
    int k=(n-1)/2;
    x[k]=p[n-1];
    for(int i=k+1;i<n;i++)x[i]=p[n-2*(i-k)];
    for(int i=k-1;i>=0;i--)x[i]=p[n-2*(k-i)-1];
    double m=0,t=0;
    for (int i=0;i<n;i++)
    {
        m+=p[i];
        for(int j=i+1;j<n;j++)t+=x[i]*x[j]*(j-i);
    }
    return t/m/m;
}
```

算法实现题 4-5 程序存储问题

★ 问题描述

设有 n 个程序$\{1,2,\cdots,n\}$要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 l_i,$1 \leqslant i \leqslant n$。

程序存储问题要求确定这 n 个程序在磁带上的一个存储方案,使得能够在磁带上存储尽可能多的程序。

★ 算法设计

对于给定的 n 个程序存放在磁带上的长度,计算磁带上最多可以存储的程序数。

★ 数据输入

由文件 input. txt 给出输入数据。第 1 行是 2 个正整数,分别表示文件个数 n 和磁带的长度 L。第 2 行中有 n 个正整数,表示程序存放在磁带上的长度。

★ 结果输出

将计算出的最多可以存储的程序数输出到文件 output. txt。

输入文件示例　　　　输出文件示例
input.txt　　　　　output.txt
6 50　　　　　　　　5
2 3 13 8 80 20

分析与解答：

贪心策略：最短程序优先。

```
public static int greedy(int []x,int m)
{
    int i=0,sum=0,n=x.length;
    QuickSort.quickSort(x,n);
    while(i<n)
    {
        sum+=x[i];
        if(sum<=m) i++;
        else return i;
    }
    return n;
}
```

算法实现题 4-6　最优服务次序问题

★ 问题描述

设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 t_i，$1 \leqslant i \leqslant n$。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小？平均等待时间等于 n 个顾客等待服务时间的总和除以 n。

★ 算法设计

对于给定的 n 个顾客需要的服务时间，计算最优服务次序。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示有 n 个顾客。第 2 行中有 n 个正整数，表示 n 个顾客需要的服务时间。

★ 结果输出

将计算出的最小平均等待时间输出到文件 output.txt。

输入文件示例　　　　输出文件示例
input.txt　　　　　output.txt
10　　　　　　　　　532.00
56 12 1 99 1000 234 33 55 99 812

分析与解答：

贪心策略：最短服务时间优先。

```
public static double greedy(int [ ]x)
{
    int n=x.length;
```

```
    QuickSort.quickSort(x,n);
    for(int i=1;i<n;i++)x[i]+=x[i-1];
    double t=0;
    for(int i=0;i<n;i++) t+=x[i];
    t/=n;
    return t;
}
```

算法实现题 4-7　多处最优服务次序问题

★ 问题描述

设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 t_i，$1 \leqslant i \leqslant n$。共有 s 处可以提供此项服务。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小？平均等待时间等于 n 个顾客等待服务时间的总和除以 n。

★ 算法设计

对于给定的 n 个顾客需要的服务时间和 s 的值，计算最优服务次序。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 s，表示有 n 个顾客且有 s 处可以提供顾客需要的服务。第 2 行中有 n 个正整数，表示 n 个顾客需要的服务时间。

★ 结果输出

将计算出的最小平均等待时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 2	336
56 12 1 99 1000 234 33 55 99 812	

分析与解答：

贪心策略：最短服务时间优先。

```
public static double greedy(int []x,int s)
{
    int []st=new int[s+1];
    int []su=new int[s+1];
    for(int i=0;i<=s;i++){st[i]=0;su[i]=0;}
    int n=x.length;
    QuickSort.quickSort(x,n);
    int i=0,j=0;double t=0;
    while(i<n)
    {
        st[j]+=x[i];
        su[j]+=st[j];
        i++;j++;
        if (j==s) j=0;
    }
    for(i=0;i<s;i++) t+=su[i];
```

```
    return t/n;
}
```

算法实现题 4-8　d 森林问题

★ 问题描述

设 T 是一棵带权树，树的每一条边带一个正权。又设 S 是 T 的顶点集，T/S 是从树 T 中将 S 中顶点删去后得到的森林。如果 T/S 中所有树的从根到叶的路长都不超过 d，则称 T/S 是一个 d 森林。

(1) 设计一个算法求 T 的最小顶点集 S，使 T/S 是 d 森林。（提示：从叶向根移动）

(2) 分析算法的正确性和计算复杂性。

(3) 设 T 中有 n 个顶点，则算法的计算时间复杂性应为 $O(n)$。

★ 算法设计

对于给定的带权树，计算最小分离集 S。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示给定的带权树有 n 个顶点，编号为 $1,2,\cdots,n$。编号为 1 的顶点是树根。接下来的 n 行中，第 $i+1$ 行描述与 i 个顶点相关联的边的信息。每行的第 1 个正整数 k 表示与该顶点相关联的边数。其后 $2k$ 个数中，每两个数表示 1 条边。第 1 个数是与该顶点相关联的另一个顶点的编号，第 2 个数是边权值。当 $k=0$ 时表示相应的结点是叶结点。文件的最后一行是正整数 d，表示森林中所有树的从根到叶的路长都不超过 d。

★ 结果输出

将计算出的最小分离集 S 的顶点数输出到文件 output.txt。如果无法得到所要求的 d 森林，则输出"No Solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
4	1
2 2 3 3 1	
1 4 2	
0	
0	
4	

分析与解答：

用父亲数组 parent 表示树；leaf 存储叶结点编号；readin 读入初始数据。

```
public static void readin()
{
    ReadStream keyboard = new ReadStream();
    n=keyboard.readInt();
    for (int i=1; i<=n; i++)
    {
        deg[i]=keyboard.readInt();
```

```
            for (int j=0;j<deg[i];j++)
            {
                p=keyboard.readInt();
                len=keyboard.readInt();
                parent[p]=i;parlen[p]=len;
            }
            if (deg[i]==0) leaf[++leaf[0]]=i;
        }
        p=keyboard.readInt();
    }
```

从叶结点向根结点移动。从根结点到叶结点的路长超过 d 时,将该子树分离。

```
public static int count()
{
    int total=0;
    for (int i=1;i<=leaf[0];i++)
        if (leaf[i]!=1) {
        //非根结点
            int plen=parlen[leaf[i]], par=parent[leaf[i]];
            if (cut[par]<1 && dist[leaf[i]]+plen>p){
                total++;
                cut[par]=1;
                par=parent[par];
        }
        else if(cut[par]<1 && dist[par]<dist[leaf[i]]+plen)dist[par]=dist[leaf[i]]+plen;
        if(--deg[par]==0) leaf[++leaf[0]]=par;
    }
    return total;
}
```

算法实现题 4-9　汽车加油问题

★ 问题描述

一辆汽车加满油后可行驶 n 千米。旅途中有若干个加油站。设计一个有效算法,指出应在哪些加油站停靠加油,使沿途加油次数最少。并证明算法能产生一个最优解。

★ 算法设计

对于给定的 n 和 k 个加油站位置,计算最少加油次数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k,表示汽车加满油后可行驶 n 公里,且旅途中有 k 个加油站。第 2 行中有 $k+1$ 个整数,表示第 k 个加油站与第 $k-1$ 个加油站之间的距离。第 0 个加油站表示出发地,汽车已加满油。第 $k+1$ 个加油站表示目的地。

★ 结果输出

将计算出的最少加油次数输出到文件 output.txt。如果无法到达目的地,则输出"No Solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
7 7	4
1 2 3 4 5 1 6 6	

分析与解答：

贪心策略：最远加油站优先。

```
public static int greedy(int []x,int n)
{
    int sum=0,k=x.length;
    for(int j=0;j<k;j++)
        if(x[j]>n)
        {
            System.out.println("No solution!");
            return -1;
        }
    for(int i=0,s=0;i<k;i++)
    {
        s+=x[i];
        if(s>n) {sum++;s=x[i];}
    }
    return sum;
}
```

算法实现题 4-10 区间覆盖问题

★ 问题描述

设 $x_1,x_2,\cdots,x_n$ 是实直线上的 n 个点。用固定长度的闭区间覆盖这 n 个点，至少需要多少个这样的固定长度闭区间？设计解此问题的有效算法，并证明算法的正确性。

★ 算法设计

对于给定的实直线上的 n 个点和闭区间的长度 k，计算覆盖点集的最少区间数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k，表示有 n 个点，且固定长度闭区间的长度为 k。第 2 行中有 n 个整数，表示 n 个点在实直线上的坐标(可能相同)。

★ 结果输出

将计算出的最少区间数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	3
1 2 3 4 5 -2 6	

分析与解答：

贪心策略：每次覆盖尽可能多的点。

```
public static int greedy(int [ ]x,int k)
{
    int sum=1,n=x.length;
    QuickSort.quickSort(x,n);
    for(int i=1,temp=x[0];i<n;i++)
        if (x[i]-temp>k){sum++;temp=x[i];}
    return sum;
}
```

算法实现题 4-11　硬币找钱问题

★ 问题描述

设有6种不同面值的硬币,各硬币的面值分别为5分、1角、2角、5角、1元、2元。现要用这些面值的硬币来购物和找钱。购物时可以使用的各种面值的硬币个数存于数组Coins[1:6]中,商店里各面值的硬币有足够多。在1次购物中希望使用最少硬币个数。

例如,1次购物需要付款0.55元,没有5角的硬币,只好用2×20+10+5共4枚硬币来付款。付出1元,找回4角5分,同样需要4枚硬币。但是如果付出1.05元(1枚1元和1枚5分),找回5角,则只需要3枚硬币。这个方案用的硬币个数最少。

★ 算法设计

对于给定的各种面值的硬币个数和付款金额,计算使用硬币个数最少的交易方案。

★ 数据输入

由文件input.txt给出输入数据。每一行有6个整数和1个有2位小数的实数,分别表示可以使用的各种面值的硬币个数和付款金额。文件以6个0结束。

★ 结果输出

将计算出的最少硬币个数输出到文件output.txt。结果应分行输出,每行一个数据。如果不可能完成交易,则输出impossible。

输入文件示例	输出文件示例
input.txt	output.txt
2 4 2 2 1 0 0.95	2
2 4 2 0 1 0 0.55	3
0 0 0 0 0 0	

分析与解答:

贪心策略:最大面值优先。

算法实现题 4-12　删数问题

★ 问题描述

给定n位正整数a,去掉其中任意$k \leq n$个数字后,剩下的数字按原次序排列组成一个新的正整数。对于给定的n位正整数a和正整数k,设计一个算法找出剩下数字组成的新数最小的删数方案。

★ 算法设计

对于给定的正整数a,计算删去k个数字后得到的最小数。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行是1个正整数 a。第2行是正整数 k。

★ 结果输出

将计算出的最小数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
178543	13
4	

分析与解答：

贪心策略：最近下降点优先。

```
public static void delek(String a)
{
    int i,m=a.length();
    if (k>=m){System.out.println(0);return;}
    while (k>0)
    {
        for(i=0; (i<a.length()-1) && (a.charAt (i)<=a.charAt (i+1));i++);
        a=a.Remove (i,1);k--;
    }
    while(a.length()>1 && a.charAt (0)=='0') a=a.Remove(0,1);
    System.out.println(a);
}
```

算法实现题 4-13　数列极差问题

★ 问题描述

在黑板上写了 N 个正数组成的一个数列，进行如下操作：每一次擦去其中两个数设为 a 和 b，然后在数列中加入一个数 $a\times b+1$，如此下去直至黑板上只剩下一个数。在所有按这种操作方式最后得到的数中，最大的数记为 max，最小的数记为 min，则该数列的极差 M 定义为 $M=\max-\min$。

★ 算法设计

对于给定的数列，计算出其极差 M。

★ 数据输入

由文件 input.txt 给出输入的数列，第1行是数列的长度 N（不超过2000）。第2行起是数列中的 N 个数，相邻两个数由空格分隔。文件名由键盘输入。

★ 结果输出

将计算出的数列极差 M 写入文件 output.txt。结果应分两行输出，第1行是数 M 的位数，第2行是数 M。

输入文件示例	输出文件示例
input.txt	output.txt
3	1
1 1 1	0

分析与解答:

贪心策略:与 Huffman 算法类似。

算法实现题 4-14　嵌套箱问题

★ 问题描述

一个 d 维箱 $(x_1,x_2,\cdots,x_d)$ 嵌入另一个 d 维箱 $(y_1,y_2,\cdots,y_d)$ 是指存在 $1,2,\cdots,d$ 的一个排列 π,使得 $x_{\pi(1)}<y_1,x_{\pi(2)}<y_2,\cdots,x_{\pi(d)}<y_d$。

(1) 证明上述箱嵌套关系具有传递性。

(2) 试设计一个有效算法,用于确定一个 d 维箱是否可嵌入另一个 d 维箱。

(3) 给定由 n 个 d 维箱组成的集合 $\{B_1,B_2,\cdots,B_n\}$,试设计一个有效算法找出这 n 个 d 维箱中的一个最长嵌套箱序列,并用 n 和 d 描述算法的计算时间复杂性。

★ 算法设计

给定由 n 个 d 维箱,试设计一个有效算法,找出这 n 个 d 维箱中的一个最长嵌套箱序列。

★ 数据输入

由文件 input.txt 提供输入数据。文件含多个测试数据项。每个测试数据项的第 1 行中有 2 个整数 n 和 d,分别表示箱的个数和维数。其后 n 行每行有 d 个正整数,表示箱的各维的长度。

★ 结果输出

对每个测试数据项,输出其最长嵌套箱序列的长度和从小到大排列的最长嵌套箱序列。所有结果输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
5 2	5
3 7	3 1 2 4 5
8 10	4
5 2	7 2 5 6
9 11	
21 18	
8 6	
5 2 20 1 30 10	
23 15 7 9 11 3	
40 50 34 24 14 4	
9 10 11 12 13 14	
31 4 18 8 27 17	
44 32 13 19 41 19	
1 2 3 4 5 6	
80 37 47 18 21 9	

分析与解答:

首先,对所给的 n 个 d 维箱 $\{B_1,B_2,\cdots,B_n\}$ 建立一个有向图 $G=(V,E)$:

$V=\{1,2,\cdots,n\}$，顶点 i 对应于 d 维箱 B_i。$E=\{(i,j)|B_i \subset B_j, j=1\sim n, i\neq j\}$。

由箱嵌套关系的传递性、反自反性和反对称性即知，所建立的图 G 是一个 DAG。对所建立的图 G，求其最长路算法即可求出最长的箱嵌套序列。算法所需的计算时间为 $O(n^2 d\log d)$。

算法实现题 4-15　套汇问题

★ 问题描述

套汇是指利用货币汇兑率的差异将一个单位的某种货币转换为大于一个单位的同种货币。例如，假定 1 美元可以买 0.7 英镑，1 英镑可以买 9.5 法郎，且 1 法郎可以买 0.16 美元。通过货币兑换，一个商人可以从 1 美元开始买入，得到 $0.7\times9.5\times0.16=1.064$ 美元，从而获得 6.4%的利润。

★ 算法设计

给定 n 种货币 $c_1,c_2,\cdots,c_n$ 的有关兑换率，试设计一个有效算法，用以确定是否存在套汇的可能性。

★ 数据输入

由文件 input.txt 提供输入数据。文件含多个测试数据项。每个测试数据项的第 1 行中只有 1 个整数 n $(1\leqslant n\leqslant 30)$，表示货币总数。其后 n 行给出 n 种货币的名称。接下来的 1 行中有 1 个整数 m，表示有 m 种不同的货币兑换率。其后 m 行给出 m 种不同的货币兑换率，每行有 3 个数据项 c_i，r_{ij} 和 c_j，表示货币 c_i 和 c_j 的兑换率为 r_{ij}。文件最后以数字 0 结束。

★ 结果输出

对每个测试数据项 j，如果存在套汇的可能性则输出 case j yes，否则输出 case j no。所有结果输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3	case 1 yes
USDollar	case 2 no
BritishPound	
FrenchFranc	
3	
USDollar 0.5 BritishPound	
BritishPound 10.0	
FrenchFranc	
FrenchFranc 0.21 USDollar	
0	

分析与解答：

通过计算兑换率矩阵的传递闭包进行判断。算法主体如下：

```
while (true)
{
    n=keyboard.readInt();
```

```
        if (n==0) break;
        for (i=0; i<n; i++)name[i]=keyboard.readString();
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)r[i][j]=0.0;
        edges=keyboard.readInt();
        for (i=0; i<edges; i++)
        {
            a=keyboard.readString();
            x=keyboard.readDouble();
            b=keyboard.readString();
            for (j=0; a.CompareTo(name[j])!=0;j++);
            for (k=0; b.CompareTo(name[k])!=0; k++);
            r[j][k]=x;
        }
        for (i=0; i<n; i++) r[i][i]=max(1.0,r[i][i]);
        for (k=0; k<n; k++)
            for (i=0; i<n; i++)
                for (j=0; j<n; j++)
                    r[i][j]=max(r[i][j],r[i][k] * r[k][j]);
        for (i=0; i<n; i++) if (r[i][i]>1.0) break;
        if (i<n)System.out.println("Case "+(++cases)+"Yes");
        else System.out.println("Case "+(++cases)+" No");
    }
```

算法实现题 4-16 信号增强装置问题

★ 问题描述

各种资源传输网络的功能是将始发地的资源通过网络传输到一个或多个目的地。例如,通过石油或者天然气输送管网可以将从油田开采的石油和天然气传送给消费者。同样,通过高压传输网络可以将发电厂生产的电力传送给用电消费者。为了使问题更具一般性,用术语信号统称网络中传输的资源(石油、天然气、电力等)。各种资源传输网络统称为信号传输网络。信号经信号传输网络传输时,需要消耗一定的能量,并导致传输能量的衰减(油压、气压、电压等)。当传输能量衰减量(压降)达到某个阈值时,将导致传输故障。为了保证传输畅通,必须在传输网络的适当位置放置信号增强装置,确保传输能量的衰减量不超过其衰减量容许值。

为了简化问题,假定给定的信号传输网络是以信号始发地为根的一棵树 T。在树 T 的每一个结点处(除根结点外)可以放置一个信号增强装置。树 T 的结点也代表传输网络的消费结点。信号经过树 T 的结点传输到其儿子结点。树的每一边上的正权是流经该边的信号所发生的信号衰减量。信号衰减量是可加的。

信号增强装置问题要求对于一个给定的信号传输网络,计算如何放置最少的信号增强装置来保证网络传输的畅通。

★ 算法设计

对于给定的带权树,计算放置信号增强装置最少数量。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有1个正整数 n，表示给定的带权树有 n 个顶点，编号为 $1,2,\cdots,n$。编号为1的顶点是树根。接下来的 n 行中，第 $i+1$ 行描述与 i 个顶点相关联的边的信息。每行的第1个正整数 k 表示与该顶点相关联的边数。其后 $2k$ 个数中，每两个数表示1条边。第1个数是与该顶点相关联的另一个顶点的编号，第2个数是边权值。文件的最后1行是正整数 d，表示衰减量容许值。

★ 结果输出

将编程计算出的最小信号增强装置数输出到文件 output.txt。如果无法得到满足要求的网络，则输出"No Solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
4	1
2 2 3 3 1	
2 1 3 4 2	
1 1 1	
1 2 2	
4	

分析与解答：

与习题4-14解法相同。

算法实现题 4-17　磁带最大利用率问题

★ 问题描述

设有 n 个程序 $\{1,2,\cdots,n\}$ 要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 l_i，$1\leqslant i\leqslant n$。

程序存储问题要求确定这 n 个程序在磁带上的一个存储方案，使得能够在磁带上存储尽可能多的程序。在保证存储最多程序的前提下还要求磁带的利用率达到最大。

★ 算法设计

对于给定的 n 个程序存放在磁带上的长度，计算磁带上最多可以存储的程序数和占用磁带的长度。

★ 数据输入

由文件 input.txt 给出输入数据。第1行是2个正整数，分别表示文件个数 n 和磁带的长度 L。第2行中有 n 个正整数，表示程序存放在磁带上的长度。

★ 结果输出

将计算出的最多可以存储的程序数和占用磁带的长度以及存放在磁带上的每个程序的长度输出到文件 output.txt。第1行输出最多可以存储的程序数和占用磁带的长度；第2行输出存放在磁带上的每个程序的长度。

输入文件示例	输出文件示例
input.txt	output.txt
9 50	5 49

2 3 13 8 80 20 21 22 23　　　　2 3 13 8 23

分析与解答：

贪心策略：最短程序优先。求得最多可以存储的程序个数 m 后，再求最大利用率。问题转化为第5章中的装载问题，但 m 已知。对第5章中的装载问题的解略作修改如下：

```
public static void maxLoading(int i)
{
    if(i>n)
    {
        if(xm==m)
        {
            for(int j=1;j<=n;j++) bestx[j]=x[j];
            bestw=cw;
        }
        return;
    }
    r-=w[i];
    if(cw+w[i]<=c && xm<m)
    {
        x[i]=1;xm++;
        cw+=w[i];
        maxLoading(i+1);
        cw-=w[i];xm--;
    }
    if(cw+r>bestw){x[i]=0; maxLoading(i+1);}
    r+=w[i];
}
```

算法实现题 4-18　非单位时间任务安排问题

★ 问题描述

具有截止时间和误时惩罚的任务安排问题可描述如下：

(1) 给定 n 个任务的集合 $S=\{1,2,\cdots,n\}$。

(2) 完成任务 i 需要 t_i 时间，$1\leqslant i\leqslant n$。

(3) 任务 i 的截止时间 d_i，$1\leqslant i\leqslant n$，即要求任务 i 在时间 d_i 之前结束。

(4) 任务 i 的误时惩罚 w_i，$1\leqslant i\leqslant n$，即任务 i 未在时间 d_i 之前结束将招致 w_i 的惩罚；若按时完成则无惩罚；

任务安排问题要求确定 S 的一个时间表(最优时间表)使得总误时惩罚达到最小。

★ 算法设计

对于给定的 n 个任务，计算总误时惩罚最小的最优时间表。

★ 数据输入

由文件 input.txt 给出输入数据。第1行是1个正整数 n，表示任务数。接下来的 n 行中，每行有3个正整数 a,b,c，表示完成相应任务需要时间 a，截止时间为 b，误时惩罚为 c。

★ 结果输出

将计算出的总误时惩罚输出到文件 output.txt。

输入文件示例	输出文件示例
input. txt	output. txt
7	110
1 4 70	
2 2 60	
1 4 50	
1 3 40	
1 1 30	
1 4 20	
3 6 80	

分析与解答：

首先将任务依其截止时间非减序排列。

设对任务 $1,2,\cdots,i$，截止时间为 d 的最小误时惩罚为 $p(i,d)$，则 $p(i,d)$ 具有最优子结构性质且满足如下递归式：

$$p(i,d)=\min\{p(i-1,d)+w(i),p(i-1,\min\{d,d_i\}-t_i)\}$$

$$p(1,d)=\begin{cases}0 & t_1\leqslant d\\ w(1) & t_1>d\end{cases}$$

据此可设计解此问题的算法如下。

init 读入数据并作初始化处理。

```
public static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    tsk=new tk[n];
    for (int i=0;i<n;i++)tsk[i]=new tk();
    for (int i=0;i<n;i++)
    {
        tsk[i].kk[0]=keyboard.readInt();
        tsk[i].kk[1]=keyboard.readInt();
        tsk[i].kk[2]=keyboard.readInt();
    }
    MergeSort.mergeSort(tsk);
    d=tsk[n-1].kk[1];
    f=new int[n][d+1];
    for(int i=0;i<n;i++)
        for(int j=0;j<=d;j++)
            f[i][j]=Integer.MAX_VALUE;
}
```

dyna 作动态规划计算。

```
public static void dyna()
{
```

```
        for(int i=0;i<=d;i++)
            if(tsk[0].kk[0]<=i)f[0][i]=0;
            else f[0][i]=tsk[0].kk[2];
        for(int i=1;i<n;i++)
        {
            for(int j=0;j<=d;j++)
            {
                f[i][j]=f[i-1][j]+tsk[i].kk[2];
                int jj=tsk[i].kk[1]>j? j:tsk[i].kk[1];
                if(jj>=tsk[i].kk[0] && f[i][j]>f[i-1][jj-tsk[i].kk[0]])
                    f[i][j]=f[i-1][jj-tsk[i].kk[0]];
            }
        }
    }
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    init();
    dyna();
    System.out.println(f[n-1][d]);
}
```

算法所需的计算时间为 $O(n\log n+nd)$。其中，$d=\max\limits_{1\leqslant i\leqslant n}\{d_i\}$。

算法实现题 4-19　多元 Huffman 编码问题

★ 问题描述

在一个操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次至少选 2 堆最多选 k 堆石子合并成新的一堆，合并的费用为新的一堆的石子数。试设计一个算法，计算出将 n 堆石子合并成一堆的最大总费用和最小总费用。

★ 算法设计

对于给定 n 堆石子，计算合并成一堆的最大总费用和最小总费用。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 n 和 k，表示有 n 堆石子，每次至少选 2 堆最多选 k 堆石子合并。第 2 行有 n 个数，分别表示每堆石子的个数。

★ 结果输出

将计算出的最大总费用和最小总费用输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	593 199
45 13 12 16 9 5 22	

分析与解答：

不妨设 $n \bmod (k-1)=1$，若不满足，可增加若干 0。

贪心策略：每次选最小的 k 个元素进行合并。与二元 Huffman 算法类似，可证明其满足贪心选择性质。具体算法描述如下：

```
public static int huffman(int a[ ], int n)
{
    MinHp H=new MinHp (1);
    H.initialize(a,n);
    int i,j,m,x,t,sum;
    m=(k-n%(k-1))%(k-1);
    for(i=1,t=0;i<=k-m;i++)
    {
        x=H.removeMin();
        t+=x;
    }
    sum=t;n=(n-k+m)/(k-1);
    H.put(t);
    for (i=1;i<=n;i++)
    {
        for(j=1,t=0;j<=k;j++)
        {
            x=H.removeMin();t+=x;
        }
        sum+=t;H.put(t);
    }
    return sum;
}
```

算法所需的计算时间为 $O(n\log_k n)$。

算法实现题 4-20　多元 Huffman 编码变形

★ 问题描述

在一个操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定在合并过程中最多可以有 $m(k)$ 次选 k 堆石子合并成新的一堆，$2\leqslant k\leqslant n$。合并的费用为新的一堆的石子数。试设计一个算法，计算出将 n 堆石子合并成一堆的最小总费用。

★ 算法设计

对于给定 n 堆石子，计算合并成一堆的最小总费用。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行有 1 个正整数 n，表示有 n 堆石子。第 2 行有 n 个数，分别表示每堆石子的个数。第 3 行有 $n-1$ 个数，分别表示 $m(k)(2\leqslant k\leqslant n)$ 的值。

★ 结果输出

将计算出的最小总费用输出到文件 output.txt 中。问题无解时输出“No solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
7	136
45 13 12 16 9 5 22	
3 3 0 2 1 0	

分析与解答：

首先找到向量 $\boldsymbol{y}$，使 $\boldsymbol{y}=\max\left\{\boldsymbol{y}\leqslant m \,\middle|\, \sum_{k=2}^{n} y(k)(k-1)=n-1\right\}$。如果找不到，则问题无解，否则用向量 $\boldsymbol{y}$ 作贪心计算。

贪心策略：每次选最小的 k，作 $y(k)$ 次 k 个元素进行合并。与二元 Huffman 算法类似，可证明其满足贪心选择性质。具体算法描述如下。

search 找向量 $\boldsymbol{y}$。

```
public static void search(int dep,int sum)
{
    if(dep==n-1) {
        if(sum==n-1) found=true;
        return;
    }
    int ii=n/(n-dep-1);
    if(ii>c[n-dep])ii=c[n-dep];
    for(int i=ii;i>=0;i--)
    {
        b[n-dep]=i;
        sum+=i*(n-dep-1);
        search(dep+1,sum);
        if(found) return;
        sum-=i*(n-dep-1);
    }
}
```

guffman 作贪心计算。

```
public static int guffman(int []a,int []b,int n)
{
    MinHp H=new MinHp (1);
    H.initialize(a,n);
    int i,j,k,x,t,sum=0;
    for (i=2;i<=n;i++)
    {
        for(k=1;k<=b[i];k++)
        {
            for(j=1,t=0;j<=i;j++){x=H.removeMin();t+=x;}
```

```
            sum+=t;H.put(t);
        }
    }
    return sum;
```

算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    a=new int[n+1];
    b=new int[n+1];
    c=new int[n+1];
    for(int i=1;i<=n;i++)a[i]=keyboard.readInt();
    for(int i=2;i<=n;i++)c[i]=keyboard.readInt();
    for(int i=0;i<=n;i++)b[i]=0;found=false;
    search(0,0);
    if(!found)System.out.println("No solution!");
    else System.out.println(guffman(a,b,n));
}
```

算法实现题 4-21　区间相交问题

★ 问题描述

给定 x 轴上 n 个闭区间。去掉尽可能少的闭区间，使剩下的闭区间都不相交。

★ 算法设计

给定 n 个闭区间，计算去掉的最少闭区间数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行是正整数 n，表示闭区间数。接下来的 n 行中，每行有 2 个整数，分别表示闭区间的 2 个数端点。

★ 结果输出

将计算出的去掉的最少闭区间数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	2
10 20	
10 15	
20 15	

分析与解答：

与活动安排问题类似，每次选取右端点坐标最小的闭区间，保留该闭区间，并将与其相交的闭区间删去。

算法实现题 4-22 任务时间表问题

★ **问题描述**

一个单位时间任务是恰好需要一个单位时间完成的任务。给定一个单位时间任务的有限集 S。关于 S 的一个时间表用于描述 S 中单位时间任务的执行次序。时间表中第 1 个任务从时间 0 开始执行直至时间 1 结束,第 2 个任务从时间 1 开始执行至时间 2 结束,……,第 n 个任务从时间 $n-1$ 开始执行直至时间 n 结束。

具有截止时间和误时惩罚的单位时间任务时间表问题可描述如下:

(1) n 个单位时间任务的集合 $S=\{1,2,\cdots,n\}$。

(2) 任务 i 的截止时间 d_i,$1\leqslant i\leqslant n$,$1\leqslant d_i\leqslant n$,即要求任务 i 在时间 d_i 之前结束。

(3) 任务 i 的误时惩罚 w_i,$1\leqslant i\leqslant n$,即任务 i 未在时间 d_i 之前结束将招致 w_i 的惩罚;若按时完成则无惩罚。

任务时间表问题要求确定 S 的一个时间表(最优时间表)使得总误时惩罚达到最小。

★ **算法设计**

给定 n 个单位时间任务,各任务的截止时间 d_i,各任务的误时惩罚 w_i,$1\leqslant i\leqslant n$,计算最优时间表。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行是正整数 n,表示任务数。接下来的 2 行中,每行有 n 个正整数,分别表示各任务的截止时间和误时惩罚。

★ **结果输出**

将计算出的最小总误时惩罚输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7	50
4 2 4 3 1 4 6	
70 60 50 40 30 20 10	

分析与解答:

见主教材,此处略。

第 5 章 回溯法

习题 5-1　装载问题改进回溯法(一)

用主教材 5.2 节中的改进策略(1)重写装载问题回溯法，使改进后算法计算时间复杂性为$O(2^n)$。

分析与解答：

先运行只计算最优值的算法 backtrack1，计算出最优装载量 bestw。由于该算法不记录最优解，故所需的计算时间为 $O(2^n)$。

```
private static void backtrack1(int i)
{
    if (i>n) {bestw=cw;return;}
    r-=w[i];
    if (cw+w[i]<=c) {cw+=w[i];backtrack1(i+1);cw-=w[i];}
    if (cw+r > bestw)backtrack1(i+1);
    r +=w[i];
}
```

然后运行改进后的算法 backtrack，在首次到达的叶结点处，即首次遇到 $i>n$ 时终止算法。由此返回的 bestx 即为最优解。

```
private static void backtrack(int i)
{
   if(found)return;
   if(i>n){
       for (int j=1;j<=n;j++)bestx[j]=x[j];
       found=true;
       return;
   }
   r-=w[i];
   if(cw+w[i]<=c){
       x[i]=1;cw+=w[i];
       backtrack(i+1);
       cw-=w[i];
   }
```

```
    if(cw+r>=bestw){x[i]=0;backtrack(i+1);}
    r+=w[i];
}
```

习题 5-2　装载问题改进回溯法(二)

用主教材5.2节中的改进策略(2)重写装载问题回溯法,使改进后算法计算时间复杂性为$O(2^n)$。

分析与解答:

在算法中动态地更新bestx。在第i层的当前结点处,当前最优解由$x[j]$,$1\leqslant j<i$和bestx$[j]$,$i\leqslant j\leqslant n$组成。每当算法回溯一层,将$x[i]$存入bestx$[i]$。这样在每个结点处更新bestx只需$O(1)$时间,从而整个算法中更新bestx所需的时间为$O(2^n)$。

```
private static void backtrack(int i)
{
    if (i>n){index=n;bestw=cw;return;}
    r -=w[i];
    if (cw+w[i]<=c){
        x[i]=1;cw+=w[i];
      backtrack(i+1);
      if (index==i){bestx[index]=1; index--;}
      cw -=w[i];
    }
    if (cw+r>bestw){
        x[i]=0;
        backtrack(i+1);
        if (index==i){bestx[index]=0; index--;}
    }
    r +=w[i];
}

public static int maxLoading(int [] ww, int cc, int [] xx)
{
    n=ww.length-1; w=ww; c=cc; cw=0; bestw=0;
    x=new int [n+1];bestx=xx; index=0;
    for (int i=1; i<=n; i++) r+=w[i];
    backtrack(1);
    return bestw;
```

习题 5-3　0-1背包问题的最优解

重写0-1背包问题的回溯法,使算法能输出最优解。

分析与解答:

为了构造最优解,必须在算法中记录与当前最优值相应的当前最优解。为此,在类Knap中增加两个私有数据成员x和bestx。x用于记录从根至当前结点的路径;bestx用于记录当前最优解。算法搜索到达叶结点处,就修正bestx的值。

修改后的算法描述如下。

backtrack 在回溯过程中记录从根至当前结点的路径。

```
private static void backtrack(int i)
{
    if (i>n){index=n;bestp=cp;return;}
    if (cw+w[i]<=c){
        x[i]=1;cw +=w[i];cp +=p[i];
        backtrack(i+1);
        if (index==i){bestx[id[i]]=1; index--;}
        cw -=w[i];cp -=p[i];
    }
    if (bound(i+1)>bestp){
        x[i]=0;
        backtrack(i+1);
        if (index==i){bestx[id[i]]=0; index--;}
    }
}
```

knapsack 作初始化，并用回溯法求解。

```
public static double knapsack(double [] pp, double [] ww, double cc, int [] xx)
{
    c=cc; n=pp.length-1; cw=0.0; cp=0.0; bestp=0.0;
    Element [] q=new Element [n];
    for (int i=1; i<=n; i++) q[i-1]=new Element(i, pp[i]/ww[i]);
    MergeSort.mergeSort(q);
    p=new double [n+1];
    w=new double [n+1];
    id=new int [n+1];
    for (int i=1; i<=n; i++){
        p[i]=pp[q[n-i].id];
        w[i]=ww[q[n-i].id];
        id[i]=q[n-i].id;
    }
    x=new int [n+1];
    bestx=xx; index=0;
    backtrack(1);
    if (bestp==0.0)for (int i=1; i<=n; i++) xx[i]=0;
    return bestp;
}
```

习题 5-4　最大团问题的迭代回溯法

试设计一个解最大团问题的迭代回溯法。

分析与解答：

与主教材中装载问题的迭代回溯法类似，最大团问题的迭代回溯法描述如下。

```
static void iterClique()
{
    for(int i=0;i<=n;i++)x[i]=0;
    int i=1;
    while(true){
        while(i<=n && ok(i)){x[i++]=1;cn++;}
        if(i>=n){
            for (int j=1;j<=n;j++)bestx[j]=x[j];
            bestn=cn;
        }
        else x[i++]=0;
        while(cn+n-i<=bestn){
            i--;
            while(i>0 && x[i]==0)i--;
            if(i==0) return;
            x[i++]=0;cn--;
        }
    }
}
```

ok 用于判断当前顶点是否可加入当前团。

```
static boolean ok(int i)
{
    for(int j=1;j<i;j++)if(x[j]>0 && a[i][j]==0) return false;
    return true;
}
```

IterClique 作初始化,并调用迭代回溯法求解。

```
public static int IterClique()
{
    cn=0;bestn=0;
    iterClique();
    return bestn;
}
```

习题 5-5　旅行售货员问题的费用上界

设 G 是有 n 个顶点的有向图,从顶点 i 发出的边的最大费用记为 $\max(i)$。

(1) 证明旅行售货员回路的费用不超过 $\sum_{i=1}^{n}\max(i)+1$。

(2) 在旅行售货员问题的回溯法中,用上面的界作为 bestc 的初始值,重写该算法,并尽可能地简化代码。

分析与解答:

(1) 任一旅行售货员回路可表示为 n 个顶点的一个排列 $(\pi(1),\pi(2),\cdots,\pi(n))$,这个回路的费用为 $h(\pi)=\sum_{i=1}^{n}a(\pi(i),\pi(i \bmod n+1))$。由此可知

$$h(\pi) - \sum_{i=1}^{n} a(\pi(i), \pi(i \bmod n + 1)) \leqslant \sum_{i=1}^{n} \max(\pi(i)) - \sum_{i=1}^{n} \max(i) < \sum_{i=1}^{n} \max(i) + 1$$

(2) 对图 G 的简单遍历即可计算出 $\sum_{i=1}^{n} \max(i) + 1$ 的值。

```
static float tsp()
{
    bestc=1;
    for(int i=1;i<=n;i++){
        float MaxCost=0;
        for(int j=1;j<=n;j++)
            if(a[i][j]<Float.MAX_VALUE && a[i][j]>MaxCost)
            MaxCost=a[i][j];
        if(MaxCost==Float.MAX_VALUE) return Float.MAX_VALUE;
        bestc+=MaxCost;
    }
    x=new int[n+1];
    for(int i=1;i<=n;i++)x[i]=i;
    cc=0;
    backtrack(2);
    return bestc;
}
```

在主教材的 TSP 回溯法中，语句 bestc==Float.MAX_VALUE 可以删去，修改如下：

```
static void backtrack(int i)
{
    if(i==n){
        if(a[x[n-1]][x[n]]<Float.MAX_VALUE  &&
            a[x[n]][1]<Float.MAX_VALUE  &&
            (cc+a[x[n-1]][x[n]]+a[x[n]][1]<bestc)){
              for(int j=1;j<=n;j++)bestx[j]=x[j];
              bestc=cc+a[x[n-1]][x[n]]+a[x[n]][1];
        }
    }
    else{
        for(int j=i;j<=n;j++)
            if(a[x[i-1]][x[j]]<Float.MAX_VALUE && (cc+a[x[i-1]][x[j]]<bestc)){
                MyMath.swap(x,i,j);
                cc +=a[x[i-1]][x[i]];
                backtrack(i+1);
                cc -=a[x[i-1]][x[i]];
                MyMath.swap(x,i,j);
            }
    }
}
```

习题 5-6　旅行售货员问题的上界函数

设 G 是有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

(1) 证明图 G 的所有前缀为 $x[1:i]$ 的旅行售货员回路的费用至少为 $\sum_{j=2}^{i} a(x_{j-1},x_j) + \sum_{j=i}^{n}\min(x_j)$，其中 $a(u,v)$ 是边 (u,v) 的费用。

(2) 利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与主教材中的算法进行比较。

分析与解答：

(1) 前缀为 $x[1:i]$ 的旅行售货员回路任一旅行售货员回路可表示为 n 个顶点的一个排列 $(x[1],x[2],\cdots,x[i],\pi(i+1),\pi(i+2),\cdots,\pi(n))$。

这个回路的费用为

$$h(\pi) = \sum_{j=2}^{i} a(x_{j-1},x_j) + a(x_i,\pi(i+1)) + \sum_{j=i+1}^{n} a(\pi(j),\pi(j \bmod n+1))$$

由此可知，

$$h(\pi) \geqslant \sum_{j=2}^{i} a(x_{j-1},x_j) + \min(x_i) + \sum_{j=i+1}^{n} \min(\pi(j))$$

$$= \sum_{j=2}^{i} a(x_{j-1},x_j) + \sum_{j=i}^{n} \min(x_j)$$

(2) 先对图 G 简单遍历，计算出 $\sum_{i=1}^{n}\min(i)$ 的值。

算法实现题 5-1　子集和问题

★ 问题描述

子集和问题的一个实例为 $\langle S,t\rangle$。其中，$S=\{x_1, x_2,\cdots, x_n\}$ 是一个正整数的集合，c 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1，使得 $\sum_{x\in S_1} x = c$。

试设计一个解子集和问题的回溯法。

★ 算法设计

对于给定的正整数的集合 $S=\{x_1, x_2,\cdots, x_n\}$ 和正整数 c，计算 S 的一个子集 S_1，使得 $\sum_{x\in S_1} x = c$。

★ 数据输入

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 c，n 表示 S 的大小，c 是子集和的目标值。第 2 行中有 n 个正整数，表示集合 S 中的元素。

★ 结果输出

将子集和问题的解输出到文件 output.txt 中。当问题无解时，输出“No solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	2 2 6
2 2 6 5 4	

分析与解答：

与装载问题类似，可设计解子集和问题的回溯法如下：

```
static boolean backtrack(int i)
{
    if (i>n) {
        for (int j=1; j<=n; j++) bestx[j]=x[j];
        bestw=cw;
        if(bestw==c)return true;
        else return false;
    }
    r-=w[i];
    if (cw+w[i]<=c) {
        x[i]=1;
        cw+=w[i];
        if (backtrack(i+1)) return true;
        cw -=w[i];}
    if (cw+r>bestw) {
        x[i]=0;
        if(backtrack(i+1))return true;}
    r +=w[i];
    return false;
}
```

算法实现题 5-2　最小长度电路板排列问题

★ 问题描述

最小长度电路板排列问题是大规模集成电路系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1,2,\cdots,n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2,\cdots,N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。

例如，设 $n=8,m=5$。给定 n 块电路板及其 m 个连接块如下：

$B=\{1,2,3,4,5,6,7,8\}$；$L=\{N_1, N_2, N_3, N_4, N_5\}$；

$N_1=\{4,5,6\}$；$N_2=\{2,3\}$；$N_3=\{1,3\}$；$N_4=\{3,6\}$；$N_5=\{7,8\}$。

这 8 块电路板的一个可能的排列如图 5-1 所示。

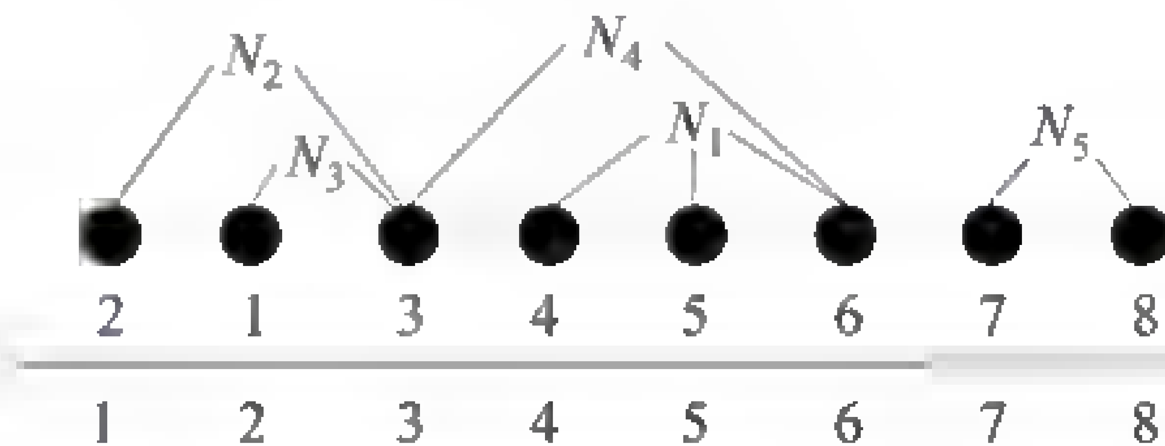


图 5-1　8 块电路板的排列

在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，在图 5-1 所示的电路板排列中，连接块 N_4 的第 1 块电路板在插槽 3 中，它的最后 1 块电路板在插槽 6 中，因此 N_4 的长度为 3。同理 N_2 的长度为 2。图中连接块最大长度为 3。试设计一个回溯法找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长

度达到最小。

★ 算法设计

对于给定的电路板连接块,设计一个算法,找出所给 n 个电路板的最佳排列,使得 m 个连接块中最大长度达到最小。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 m(其中 $1 \leqslant m, n \leqslant 20$)。接下来的 n 行中,每行有 m 个数。第 k 行的第 j 个数为0表示电路板 k 不在连接块 j 中,1表示电路板 k 在连接块 j 中。

★ 结果输出

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第1行是最小长度;第2行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答:

与主教材中电路板排列问题类似,可设计解最小长度电路板排列问题的回溯法如下。主要区别是计算连接块的长度,由算法 len 完成。

```
static int len(int ii)
{
    for (int i=1; i<=m; i++) {high[i]=0;low[i]=n+1;}
    for (int i=1; i<=ii; i++)
        for (int k=1; k<=m; k++)
            if(B[x[i]][k]>0){
                if(i<low[k]) low[k]=i;
                if(i>high[k]) high[k]=i;
            }
    int tmp=0;
    for (int k=1; k<=m; k++)
        if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k])tmp=high[k]-low[k];
    return tmp;
}
```

回溯法实体是 backtrack。

```
static void backtrack(int i)
{
    if (i==n) {
        int tmp=len(i);
        if(tmp<bestd){bestd=tmp;for (int j=1;j<=n;j ++) bestx[j]=x[j];}
    }
    else
        for (int j=i; j<=n; j++) {
            MyMath.swap(x,i,j);
            int ld =len(i);
            if (ld<bestd) backtrack(i+1);
            MyMath.swap(x,i,j);
        }
}
```

最后由 arrange 完成计算。

```
public static int arrange(int [][]BB, int nn, int mm, int []bestxx)
{
    n=nn;m=mm;
    x=new int[n+1];
    low=new int[m+1];
    high=new int[m+1];
    B=BB;bestx=bestxx; bestd=n+1;
    for (int i=1; i<=n; i++) x[i]=i;
    backtrack(1);
    return bestd;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int m=keyboard.readInt();
    int []p =new int[n+1];
    int [][]B =new int[n+1][m+1];
    for (int i =1; i<=n; i++)
        for (int j=1; j<=m; j++) B[i][j]=keyboard.readInt();
    System.out.println(arrange(B, n, m, p));
    for (int i=1; i<=n; i++) System.out.print(p[i]+" ");
    System.out.println();
}
```

算法实现题 5-3 最小重量机器设计问题

★ 问题描述

设某一机器由 n 个部件组成,每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量,c_{ij} 是相应的价格。

试设计一个算法,给出总价格不超过 c 的最小重量机器设计。

★ 算法设计

对于给定的机器部件重量和机器部件价格,计算总价格不超过 d 的最小重量机器设计。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n,m 和 d。接下来的 $2n$ 行,每行 n 个数。前 n 行是 c,后 n 行是 w。

★ 结果输出

将计算出的最小重量,以及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

分析与解答:

与背包问题类似,可设计解最小重量机器设计问题的回溯法如下:

```
static boolean backtrack(int i)
{
    if (i>n) {
        bestw=cw;
        for(int j=1;j<=n;j++)bestx[j]=x[j];
        return true;
    }
    boolean found=false;
    if(bestw<=cc)found=true;
    for(int j=1;j<=m;j++){
        x[i]=j;cw+=w[i][j];cp+=c[i][j];
        if (cp<=cc && cw<bestw) if(backtrack(i+1)) found=true;
        cw -=w[i][j];cp -=c[i][j];
    }
    return found;
}
```

算法实现题 5-4　运动员最佳匹配问题

★ 问题描述

羽毛球队有男女运动员各 n 人。给定两个 $n\times n$ 矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$。$P[i][j]$是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势；$Q[i][j]$是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响，$P[i][j]$不一定等于 $Q[j][i]$。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j]\times Q[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

★ 算法设计

设计一个算法，对于给定的男女运动员竞赛优势，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（其中 $1\leqslant n\leqslant 20$）。接下来的 $2n$ 行，每行 n 个数。前 n 行是 p，后 n 行是 q。

★ 结果输出

将计算出的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	
3 4 5	
2 2 2	
3 5 3	
4 5 1	

分析与解答：

此题的解空间显然是一棵排列树，可以套用搜索排列树的回溯法框架。

```
static void backtrack(int t)
{
    if (t>n) compute();
    else
        for (int j=t; j<=n; j++) {
            MyMath.swap(r,t,j);
            backtrack(t+1);
            MyMath.swap(r,t,j);
        }
}
```

其中，compute 计算当前配对的竞赛优势的总和。

```
static void compute()
{
    int temp=0;
    for (int i=1;i<=n;i++) temp+=p[i][r[i]]*q[r[i]][i];
    if (temp>best){
        best=temp;
        for(int i=1;i<=n;i++) bestr[i]=r[i];
    }
}
```

算法实现题 5-5　无分隔符字典问题

★ 问题描述

设 $\Sigma=\{a_1,a_2,\cdots,a_n\}$ 是 n 个互不相同的符号组成的符号集。

$L_k=\{\beta_1\beta_2\cdots\beta_k \mid \beta_i\in\Sigma,1\leqslant i\leqslant k\}$ 是 Σ 中字符组成的长度为 k 的全体字符串。

$S\subseteq L_k$ 是 L_k 的无分隔符字典，是指对任意 $a_1a_2\cdots a_k\in S$ 和 $b_1b_2\cdots b_k\in S$，有

$$\{a_2a_3\cdots a_kb_1,a_3a_4\cdots b_1b_2,\cdots,a_kb_1b_2\cdots b_{k-1}\}\cap S=\varnothing$$

无分隔符字典问题要求对给定的 n 和 Σ 以及正整数 k，编程计算 L_k 的最大无分隔符字典。

★ 算法设计

设计一个算法，对于给定的正整数 n 和 k，计算 L_k 的最大无分隔符字典。

★ 数据输入

由文件 input.txt 给出输入数据。文件第 1 行有 2 个正整数 n 和 k。

★ 结果输出

将计算出的 L_k 的最大无分隔符字典的元素个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 2	2

分析与解答：

用逐步加深的回溯法搜索解空间。

```
static void search(int dep)
{
    if(dep>lk) {
        if(s.size()>best) {best=s.size();out();}
        return;
    }
    if(oka(dep)){
        insert(dep);
        search(dep+1);
        erase(dep);
    }
    search(dep+1);
}
```

程序中的oka(dep)判断当前字符串dep是否可加入字典。本题中将字符串 $a_1a_2\cdots a_k$ 看作 k 位 n 进制数。当前字典中的字符串存储在集合 s 中。

```
static boolean oka(int b)
{
    Iterator it=s.iterator();
    while(it.hasNext()){int a=((Integer)it.next ()).intValue();if (pref(a,b)) return false;}
    return true;
}
```

pref(a,b)用于判断字符串a和b是否互不为前缀。

```
static boolean pref(int a,int b)
{
    int x=a,y=b/n;
    for(int i=0;i<k-1;i++){ak[k-i-2]=x%n;x/=n;ak[2*k-i-3]=y%n;y/=n;}
    for(int i=1;i<k;i++)  if(s.get(new Integer(digi(i)))!=null) return true;
    x=b;y=a/n;
    for(int i=0;i<k-1;i++){ak[k-i-2]=x%n;x/=n;ak[2*k-i-3]=y%n;y/=n;}
    for(int i=1;i<k;i++) if(s.get(new Integer(digi(i)))!=null) return true;
    return false;
}
```

digi将相应字符串转换为 n 进制数。

```
static int digi(int i)
{
    int ii=k+i-2;
    int x=ak[ii--];
    for(int j=0;j<k-1;j++){x*=n;x+=ak[ii];ii--;}
    return x;
}
```

readin读入数据。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    k=keyboard.readInt();
    ak=new int[2*k];
    lk=n;
    for(int i=1;i<k;i++) lk*=n;
    lk--;best=0;
}
```

实现算法的主函数如下：

```
public static void main(String [ ] args)
{
    readin();
    if(k<3)System.out.println(n);
    else{search(0);System.out.println(best);}
}
```

算法实现题 5-6　无和集问题

★ 问题描述

设 S 是正整数的集合，当且仅当 $x,y\in S$ 蕴含 $x+y\notin S$，S 是一个无和集。

对于任意正整数 k，如果可将 $\{1,2,\cdots,k\}$ 划分为 n 个无和子集 $S_1,S_2,\cdots,S_n$，称正整数 k 是 n 可分的。记 $F(n)=\max\{k\mid k$ 是 n 可分的$\}$。

试设计一个算法，对任意给定的 n，计算 $F(n)$ 的值。

★ 算法设计

对任意给定的 n，计算 $F(n)$ 的值。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

★ 结果输出

将计算出的 $F(n)$ 的值以及 $\{1,2,\cdots,F(n)\}$ 的一个 n 划分输出到文件 output.txt。文件的第 1 行是 $F(n)$ 的值。接下来的 n 行，每行是一个无和子集 S_i。

输入文件示例	输出文件示例
input.txt	output.txt
2	8
	1 2 4 8
	3 5 6 7

分析与解答：

此题是子集选取问题，其解空间显然是一棵子集树，可以套用搜索子集树的回溯法框架。

由于搜索空间很大，用搜索时间控制搜索深度。

```
static boolean search(int dep)
{
    t1=System.currentTimeMillis();
    elapsed +=(t1-t0)/1000.0;
    t0=t1;
    if(elapsed>15.0) return false;
    if(dep>k) {out();return true;}
    for(int i=1;i<=n;i++){
        if (sum[i][dep]==0){
            t[dep]=i;s[i][dep]=true;
            for(int j=1;j<dep;j++)if(s[i][j]) sum[i][dep+j]++;
            if (search(dep+1)) return true;
```

```
                s[i][dep]=false;t[dep]=0;
                for(int j=1;j<dep;j++)if(s[i][j]) sum[i][dep+j]--;
            }
        }
        return false;
}
```

算法实现题 5-7　n 色方柱问题

★ 问题描述

设有 n 个立方体，每个立方体的每一面用红、黄、蓝、绿等 n 种颜色之一染色。要把这 n 个立方体叠成一个方形柱体，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。试设计一个回溯算法，计算出 n 个立方体的一种满足要求的叠置方案。

★ 算法设计

对于给定的 n 个立方体以及每个立方体各面的颜色，计算出 n 个立方体的一种叠置方案，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，$0<n<27$，表示给定的立方体个数和颜色数均为 n。第 2 行是 n 个大写英文字母组成的字符串。该字符串的第 $k(0\leqslant k<n)$ 个字符代表第 k 种颜色。接下来的 n 行中，每行有 6 个数，表示立方体各面的颜色。立方体各面的编号如图 5-2 所示。

图 5-2 中，F 表示前面，B 表示背面，L 表示左面，R 表示右面，T 表示顶面，D 表示底面。相应地，2 表示前面，3 表示背面，0 表示左面，1 表示右面，5 表示顶面，4 表示底面。

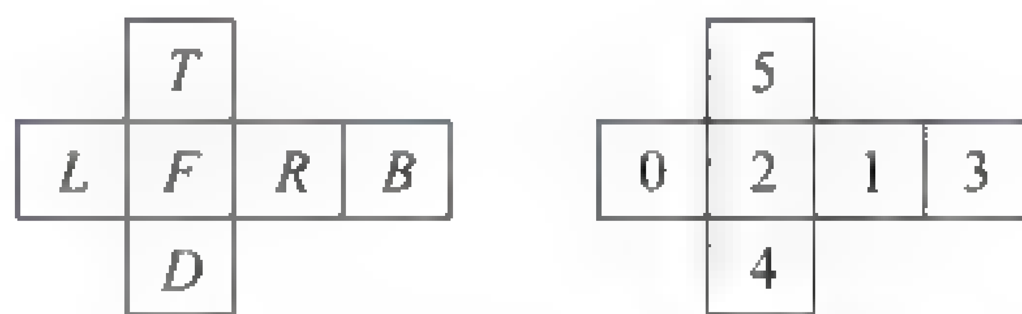


图 5-2　立方体各面的编号

例如，在示例输出文件中，第 3 行的 6 个数 0，2，1，3，0，0 分别表示第 1 个立方体的左面的颜色为 R，右面的颜色为 B，前面的颜色为 G，背面的颜色为 Y，底面的颜色为 R，顶面的颜色为 R。

★ 结果输出

将计算出的 n 个立方体的一种可行的叠置方案输出到文件 output.txt。每行 6 个字符，表示立方体各面的颜色。如果不存在所要求的叠置方案，则输出"No solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
4	RBGYRR
RGBY	YRBGRG
0 2 1 3 0 0	BGRBGY
3 0 2 1 0 1	GYYRBB
2 1 0 2 1 3	
1 3 3 0 2 2	

分析与解答：

1）算法思想

每个立方体可以按3个方向旋转，每个方向有4个不同的面，因此每个立方体可有64种不同状态。用回溯法对 n 个立方体的每种状态进行搜索，可以找到满足要求的叠置方案。然而，这样做的计算量较大。下面讨论用图论的方法进行简化。在此问题中，立方体的每对相对的面的颜色是要考查的关键因素。将每个立方体表示为有 n 个顶点的图。图中每个顶点表示一种颜色。在立方体每对相对面的顶点间连一条边。例如，图5-3(b)是图5-3(a)所示的4个立方体所相应的图。

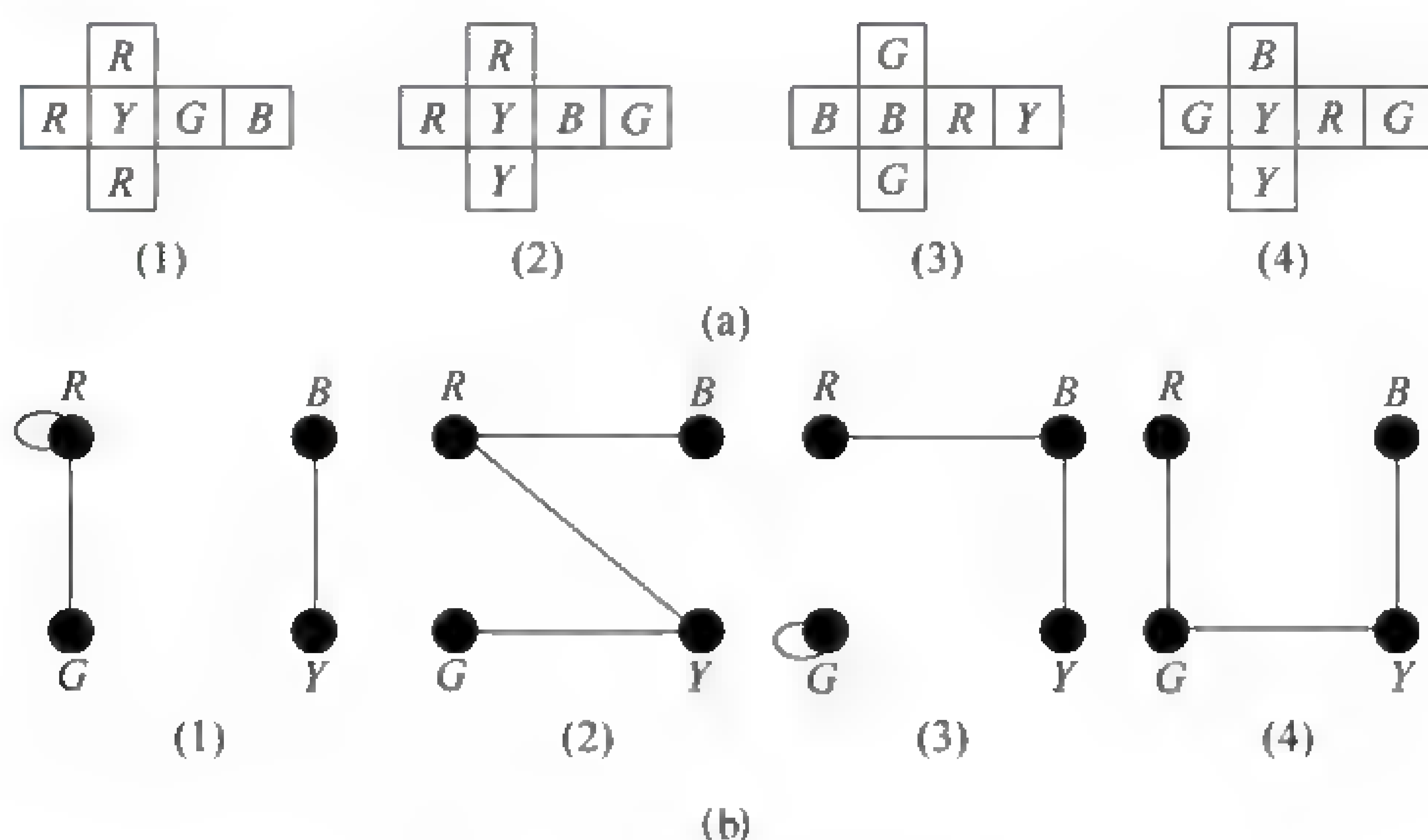


图5-3 立方体及其相应的子图

将上述子图合并，并标明每一条边来自哪一个立方体，如图5-4所示。

下一步在构成的图中，找出两个特殊子图。一个子图表示叠置的 n 个立方体的前侧面与背侧面，另一子图表示叠置的 n 个立方体的左侧面与右侧面。这两个子图应满足下述性质：

① 每个子图有 n 条边，且每个立方体恰好一条边。

② 两个子图没有公共边。

③ 子图中每个顶点的度均为2。

对于图5-4中的图，找出满足要求的两个子图，如图5-5所示。

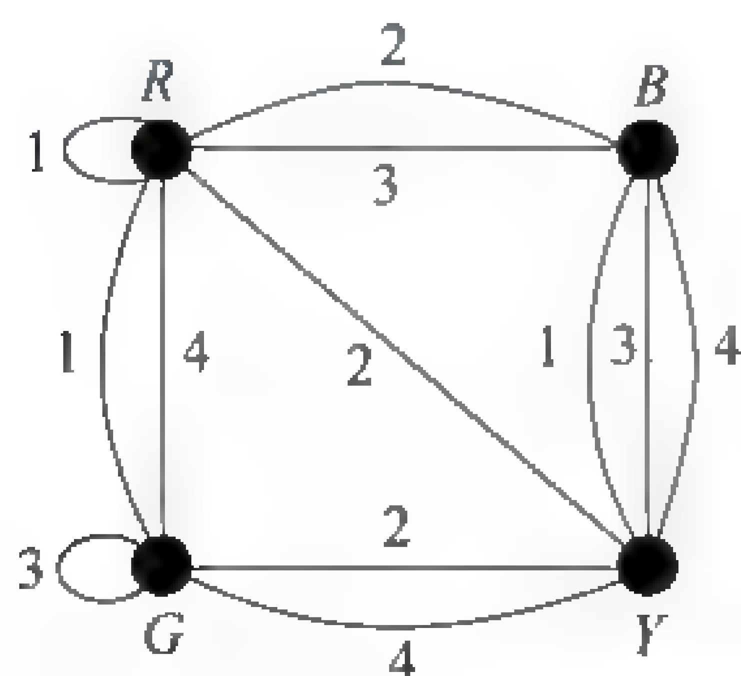


图5-4 n 个立方体及其相应的图

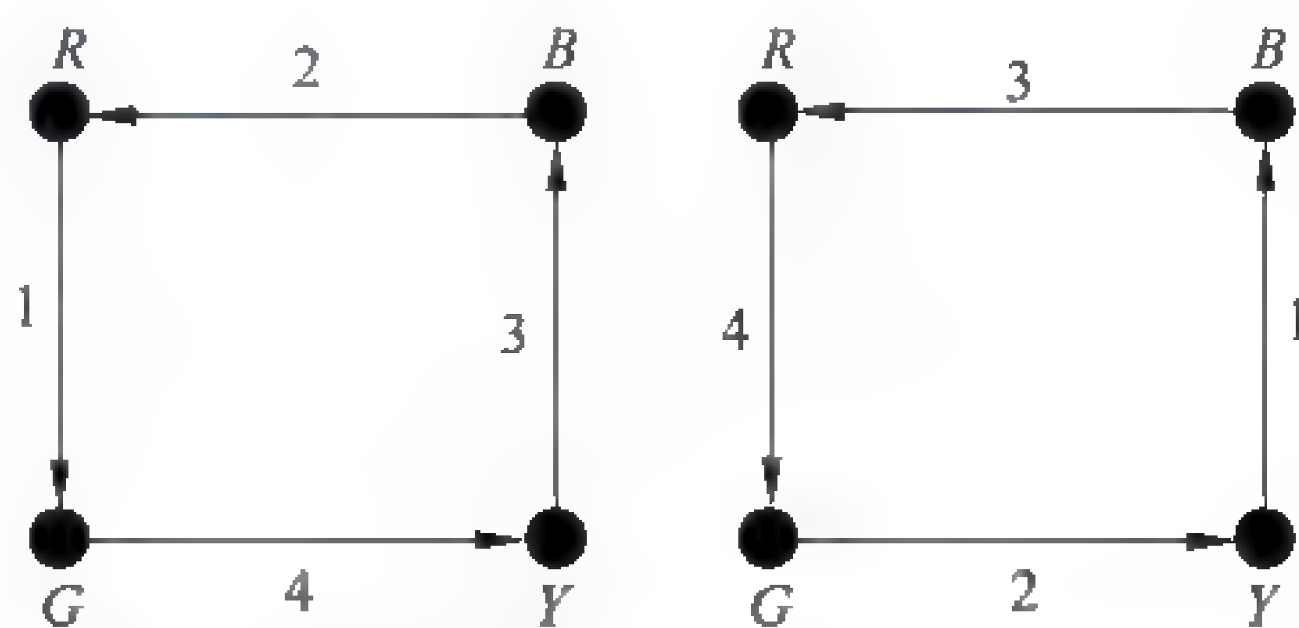


图5-5 表示 n 个立方体4个侧面的子图

给子图的每条边一个方向，使每个顶点有一条出边和一条入边。有向边的始点对应于前面和左面；有向边的终点对应于背面和右面。图5-5给出了满足要求的解如表5-1所示。

表5-1 满足4角立方体的解

立方体 \ 面	0(L)	1(R)	2(F)	3(B)
cube1	Y	B	G	R
cube2	G	Y	R	B
cube3	B	R	B	Y
cube4	R	G	Y	G

上述算法的关键是找满足性质①，②和③的子图。用回溯法。

2）算法实现

二维数组board[n][6]存储n个立方体各面的颜色。solu[n][6]存储解。

找满足性质①，②和③的子图的回溯法如下：

```
static void search()
{
    int i,t,cube;
    boolean ok,newg,over;
    int []vert=new int[n];
    int []edge=new int[n * 2];
    for (i=0;i<n;i++)vert[i]=0;
    t=-1;newg=true;
    while(t>-2){
        t++;
        cube=t%n;                                   // 每个立方体找2次
        if(newg)edge[t]=-1;
        over=false;ok=false;
        while(!ok && !over){
            edge[t]++;
            if(edge[t]>2)over=true;                 // 每个立方体只有3条边
            else ok=(t<n || edge[t]!=edge[cube]);   // 是否已用过
        }
        if(!over){
            if(++vert[board[cube][edge[t] * 2]]>2+t/n * 2)ok=false;
            if(++vert[board[cube][edge[t] * 2+1]]>2+t/n * 2)ok=false;
            if (t%n==n-1 && ok)
                for (i=0;i<n;i++)if(vert[i]>2+t/n * 2)ok=false;
            if(ok){
                if (t==n * 2-1){// 找到解
                    ans++;
                    out(edge);
                    return;
                }
                else newg=true;
            }//ok
            else{// 取下一条边
                vert[board[cube][edge[t] * 2]];
```

```
                    vert[board[cube][edge[t]*2+1]];
                t    ;newg=false;
            }
        }//over
        else{// 回溯
            t--;
            if(t>-1){
                cube=t%n;
                --vert[board[cube][edge[t]*2]];
                --vert[board[cube][edge[t]*2+1]];
            }
            t--;newg=false;
        }
    }
}
```

找到一个解后由 out 输出。

```
static void out(int edge[])
{
    int i,j,k,a,b,c,d;
    for(i=0; i<2; i++){
        for (j=0;j<n;j++) used[j]=0;
        do{
            j=0;                          // 找下一条未用边
            d=c=-1;
            while (j<n && used[j]>0) j++;
            if(j<n)
                do{
                    a=board[j][edge[i*n+j]*2];
                    b=board[j][edge[i*n+j]*2+1];
                    if (b==d){k=a;a=b;b=k;}
                    solu[j][i*2]=a;
                    solu[j][i*2+1]=b;
                    used[j]=1;
                    if (c<0) c=a;         // 开始顶点
                    d=b;
                    for (k=0; k<n; k++)  // 找下一个立方体
                        if (used[k]==0 && (board[k][edge[i*n+k]*2]==b||
                            board[k][edge[i*n+k]*2+1]==b)) j=k;
                }while(b!=c);
        }while(j<n);
    }
    for (j=0; j<n; j++){
        k=3  edge[j]  edge[j+n];
```

```
            a=board[j][k*2];
            b=board[j][k*2+1];
            solu[j][4]=a;
            solu[j][5]=b;
        }
        for (i=0; i<n; i++){
            for (j=0; j<6; j++)
                System.out.print(color[solu[i][j]]);
            System.out.println();
        }
}
```

执行算法的主函数如下：

```
public static void main(String [] args)
{
    readin();
    search();
    if(ans==0)System.out.println("No solution!");
}
```

初始数据由 readin 读入。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    board=new int[n][6];
    solu=new int[n][6];
    color=new char[n];
    used=new int[n];
    char dm=keyboard.readChar();
    for(int j=0;j<n;j++)color[j]=keyboard.readChar();
    dm=keyboard.readChar();
    for (int i=0;i<n;i++)
        for (int j=0;j<6;j++)board[i][j]=keyboard.readInt();
}
```

算法实现题 5-8　整数变换问题

★ 问题描述

整数变换问题。关于整数 i 的变换 f 和 g 定义如下：$f(i)=3i$；$g(i)=\lfloor i/2 \rfloor$。

试设计一个算法，对于给定的 2 个整数 n 和 m，用最少的 f 和 g 变换次数将 n 变换为 m。

例如，可以将整数 15 用 4 次变换将它变换为整数 4：$4=gfgg(15)$。当整数 n 不可能变换为整数 m 时，算法应如何处理？

★ **算法设计**

对任意给定的整数 n 和 m,计算将整数 n 变换为整数 m 所需要的最少变换次数。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 m。

★ **结果输出**

将计算出的最少变换次数以及相应的变换序列输出到文件 output.txt。文件的第1行是最少变换次数。文件的第2行是相应的变换序列。

输入文件示例	输出文件示例
input.txt	output.txt
15 4	4
	gfgg

分析与解答:

此题是 $3n+1$ 问题的变形。为了找最短变换序列,用逐步加深的回溯搜索。

```
static void compute()
{
    k=1;
    while(!search(1,n)) {
        k++;
        if (k>maxdep) break;
        init();
    }
    if (found) output();
    else System.out.println("No Solution!");
}
```

search 实现回溯搜索。

```
static boolean search(int dep,int n)
{
    if(dep>k) return false;
    for(int i=0;i<2;i++){
        int n1=f(n,i);t[dep]=i;
        if(n1==m || search(dep+1,n1)) {found=true;out();return true;}
    }
    return false;
}
```

算法实现题 5-9　拉丁矩阵问题

★ **问题描述**

现有 n 种不同形状的宝石,每种宝石有足够多颗。欲将这些宝石排列成 m 行 n 列的一个矩阵,$m \leqslant n$,使矩阵中每一行和每一列的宝石都没有相同形状。试设计一个算法,计算出对于给定的 m 和 n 有多少种不同的宝石排列方案。

★ 算法设计

对于给定的 m 和 n，计算出不同的宝石排列方案数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n，$0<m\leqslant n<9$。

★ 结果输出

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3	12

分析与解答：

1) 算法思想

设 n 种宝石编号为 $1,2,\cdots,n$。宝石矩阵的第 1 行从左到右排列为 $1,2,\cdots,n$，且第 1 列从上到下排列为 $1,2,\cdots,m$ 的阵列为标准拉丁矩阵。设 m 行 n 列的标准拉丁矩阵个数为 $L(m,n)$，一般情况下 m 行 n 列的拉丁矩阵个数为 $R(m,n)$。本题要求 $R(m,n)$。

容易证明，$R(m,n)=n!(n-1)!L(m,n)/(n-m)!$。于是问题可转化为求标准拉丁矩阵个数 $L(m,n)$。问题显然与排列有关，可用主教材中的排列树回溯法框架求解。

2) 算法实现

二维数组 board[m][n]存储宝石矩阵。每行初始化为单位排列，第 1 列从上到下排列为 $1,2,\cdots,m$。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    m=keyboard.readInt();
    n=keyboard.readInt();
    for(int i=1;i<=n;++i)
        for(int j=1;j<=n;++j) board[i][j]=j;
    for(int i=2;i<=n;++i)MyMath.swap(board[i],1,i);
    if(m==n)m--;
}
```

对数组 board 从上到下，从左到右递归搜索。

```
static void backtrack(int r,int c)
{
    for(int i=c;i<=n;i++)
        if(ok(r,c,board[r][i])){
            MyMath.swap(board[r],c,i);
            if(c==n){
                if(r==m)count+=1.0;
                else backtrack(r+1,2);
            }
            else backtrack(r,c+1);
```

```
                MyMath.swap(board[r],c,i);
            }
    }
```

其中,ok 用于判断在当前列中宝石是否重复。

```
    static boolean ok(int r,int c,int k)
    {
        for (int i=1;i<r;i++)if(board[i][c]==k)return false;
        return true;
    }
```

执行算法的主函数如下:

```
    public static void main(String [] args)
    {
        init();
        backtrack(2,2);
        outlong((int)count);
    }
```

其中,outlong 按公式 $R(m,n)$ 计算输出 $R(m,n)=n!(n-1)!L(m,n)/(n-m)!$的值。由于输出的值较大,这一步需要高精度计算。

注意到,当 $m=n$ 时,第 $n-1$ 行排定后,第 n 行就已确定,无须回溯。这就是 init 中的语句 if(m==n)m——的含义。当然还有其他的优化方法。

算法实现题 5-10　排列宝石问题

★ 问题描述

现有 n 种不同形状的宝石,每种 n 颗,共 n^2 颗。同一种形状的 n 颗宝石分别具有 n 种不同的颜色 $c_1,c_2,\cdots,c_n$ 中的一种颜色。欲将这 n^2 颗宝石排列成 n 行 n 列的一个方阵,使方阵中每一行和每一列的宝石都有 n 种不同形状和 n 种不同颜色。试设计一个算法,计算出对于给定的 n 有多少种不同的宝石排列方案。

★ 算法设计

对于给定的 n,计算出不同的宝石排列方案数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n,$0<n<9$。

★ 结果输出

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
1	1

分析与解答:

1) 算法思想

此题与上一题类似,用回溯法时,需对形状和颜色两种因素循环考查。

2）算法实现

二维数组 $a[n][n]$、$b[n][n]$ 分别存储宝石形状矩阵和颜色矩阵。每行初始化为单位排列。二维数组 $cc[n][n]$ 的单元 $cc[i][j]$ 用于搜索时记录形状为 i，颜色为 j 的宝石是否已用过，初始化为 false。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++) {a[i][j]=j;b[i][j]=j;cc[i][j]=false;}
}
```

对数组 a 和 b 从上到下，从左到右递归搜索。

```
static void backtrack(int r,int c)
{
    for(int i=c;i<=n;i++)
        if(ok(r,c,i,false)){
            MyMath.swap(a[r],c,i);
            for(int j=c;j<=n;j++)
                if(ok(r,c,j,true)){
                    MyMath.swap(b[r],c,j);
                    cc[a[r][c]][b[r][c]]=true;
                    if(c==n){
                        if(r==n)count+=1.0;
                        else backtrack(r+1,1);
                    }
                    else backtrack(r,c+1);
                    cc[a[r][c]][b[r][c]]=false;
                    MyMath.swap(b[r],c,j);
                }
            MyMath.swap(a[r],c,i);
        }
}
```

其中，ok 用于判断在当前列中宝石的形状和颜色是否重复。

```
static boolean ok(int r,int c,int k,boolean fla)
{
    if(fla){
        if(cc[a[r][c]][b[r][k]])return false;
        for(int i=1;i<r;i++)if(b[i][c]==b[r][k])return false;
    }
    else for(int i=1;i<r;i++)if(a[i][c]==a[r][k])return false;
    return true;
}
```

执行算法的主函数如下：

```
public static void main(String [ ] args)
{
    init();
    backtrack(1,1);
    System.out.println((int)count);
}
```

与上一题一样，注意到，第 $n-1$ 行排定后，第 n 行就已确定，无须回溯，但必须判断是否矛盾。这个任务可由 last 完成如下：

```
static boolean last()
{
    for (int j=1;j<=n;j++){
        for (int i=1;i<=n;i++) {dd[i][0]=0;dd[i][1]=0;}
        for (int i=1;i<n;i++) {dd[a[i][j]][0]=1;dd[b[i][j]][1]=1;}
        for (int i=1;i<=n;i++) {if(dd[i][0]==0)ee[j][0]=i;if(dd[i][1]==0)ee[j][1]=i;}
    }
    for(int i=1;i<=n;i++)if(cc[ee[i][0]][ee[i][1]])return false;
    return true;
}
```

最后，将回溯法中的语句 if(r==n)count+=1.0;换成 if(r==n-1){if(last())count+=1.0;}。

算法实现题 5-11　重复拉丁矩阵问题

★ 问题描述

现有 k 种不同价值的宝石，每种宝石都有足够多颗。欲将这些宝石排列成一个 m 行 n 列的矩阵，$m \leqslant n$，使矩阵中每一行和每一列的同一种宝石数都不超过规定的数量。另外还规定，宝石阵列的第 1 行从左到右和第 1 列从上到下的宝石，按宝石的价值最小字典序从小到大排列。试设计一个算法，对于给定的 k，m 和 n 以及每种宝石的规定数量，计算出有多少种不同的宝石排列方案。

★ 算法设计

对于给定的 m，n 和 k，以及每种宝石的规定数量，计算出不同的宝石排列方案数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 m，n 和 k，$0<m\leqslant n<9$。第 2 行有 k 个数，第 j 个数表示第 j 种宝石在矩阵的每行和每列出现的最多次数。这 k 个数按照宝石的价值从小到大排列。设这 k 个数为 $1\leqslant v_1\leqslant v_2\leqslant\cdots\leqslant v_k$，则 $v_1+v_2+\cdots+v_k=n$。

★ 结果输出

将计算出的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 7 3 2 2 3	84309

分析与解答：

1）算法思想

此题与前两题类似，用回溯法时，需考虑相同价值的情况。

2）算法实现

二维数组 board[m][n]存储宝石矩阵。每行初始化为单位排列，第1列从上到下排列为1，2，…，m。用数组 mv 记录规定的每种宝石的重复数，mu 记录 n 个宝石中，每个宝石的价值序号。由 init 初始化各数组。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    m=keyboard.readInt();
    n=keyboard.readInt();
    mm=keyboard.readInt();
    for(int k=1,j=1,t=0;k<=mm;k++){
        t=keyboard.readInt();
        mv[k]=t;
        while(t>0){mu[j++]=k;t--;}
    }
    for(int i=1;i<=n;++i)
        for(int j=1;j<=n;++j) board[i][j]=j;
    for(int i=2;i<=n;++i)MyMath.swap(board[i],1,i);
}
```

对数组 board 从上到下，从左到右递归搜索。

```
static void backtrack(int r,int c)
{
    for(int i=c;i<=n;i++)
        if(ok(r,c,i)){
            MyMath.swap(board[r],c,i);
            if(c==n){
                if(r==m)count+=1.0;
                else backtrack(r+1,2);
            }
            else backtrack(r,c+1);
            MyMath.swap(board[r],c,i);
        }
}
```

其中，ok 用于判断在当前列中宝石是否超过规定数。

```
static boolean ok(int r,int c,int s)
{
    int k=board[r][s];
```

```
    if(s>c) for(int t=c;t<s;t++)if(mu[board[r][t]]==mu[k])return false;
    int j=0;
    for (int i=1;i<r;i++)if(mu[board[i][c]]==mu[k])j++;
    if(j>mv[mu[k]]-1)return false;
    else return true;
}
```

执行算法的主函数如下：

```
public static void main(String [] args)
{
    init();
    backtrack(2,2);
    System.out.println((int)count);
}
```

算法实现题 5-12　罗密欧与朱丽叶的迷宫问题

★ 问题描述

罗密欧与朱丽叶的迷宫。罗密欧与朱丽叶身处一个 $m\times n$ 的迷宫中，如图 5-6 所示。每一个方格表示迷宫中的一个房间。这 $m\times n$ 个房间中有一些房间是封闭的，不允许任何人进入。在迷宫中任何位置均可沿 8 个方向进入未封闭的房间。罗密欧位于迷宫的 (p,q) 方格中，他必须找出一条通向朱丽叶所在的 (r,s) 方格的路。在到达朱丽叶之前，他必须走遍所有未封闭的房间各一次，而且要使到达朱丽叶的转弯次数为最少。每改变一次前进方向算作转弯一次。请设计一个算法帮助罗密欧找出这样一条道路。

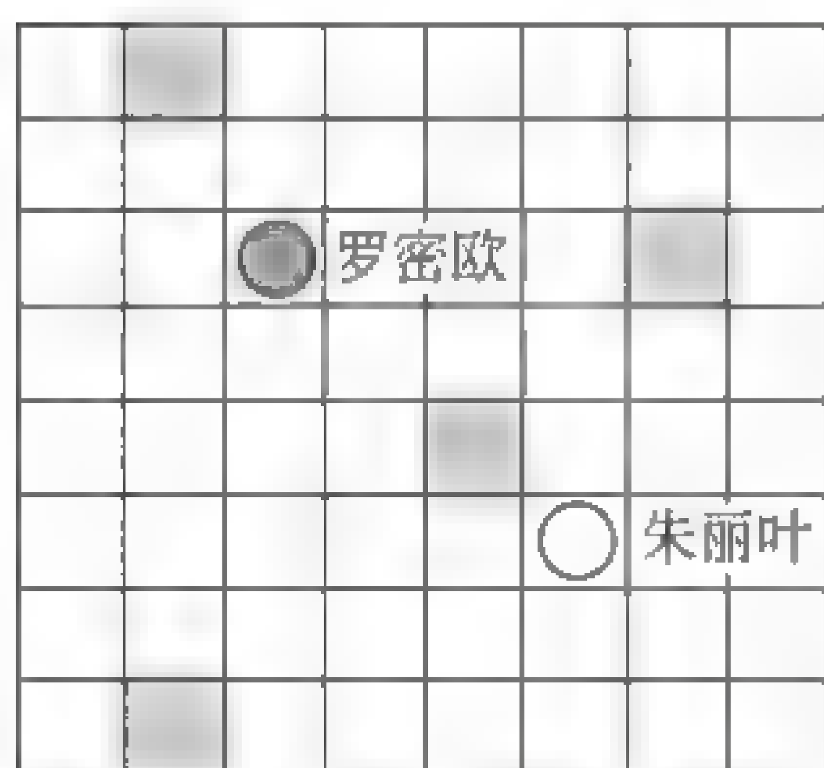


图 5-6　罗密欧与朱丽叶的迷宫

★ 算法设计

对于给定的罗密欧与朱丽叶的迷宫，计算罗密欧通向朱丽叶的所有最少转弯道路。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n,m,k，分别表示迷宫的行数，列数和封闭的房间数。接下来的 k 行中，每行 2 个正整数，表示被封闭的房间所在的行号和列号。最后的 2 行，每行也有 2 个正整数，分别表示罗密欧所处的方格 (p,q) 和朱丽叶所处的方格 (r,s)。

★ 结果输出

将计算出的罗密欧通向朱丽叶的最少转弯次数和有多少条不同的最少转弯道路输出到文件 output.txt。文件的第 1 行是最少转弯次数。文件的第 2 行是不同的最少转弯道路数。接下来的 n 行每行 m 个数，表示迷宫的一条最少转弯道路。$A[i][j]=k$ 表示第 k 步到达方格 (i,j)；$A[i][j]=-1$ 表示方格 (i,j) 是封闭的。

如果罗密欧无法通向朱丽叶，则输出“No Solution!”。

输入文件示例	输出文件示例
input. txt	output. txt
3 4 2	6
1 2	7
3 4	1 −1 9 8
1 1	2 10 6 7
2 2	3 4 5 −1

分析与解答：

在当前位置按照8个方向搜索。

```
static void search(int dep,int x,int y,int di)
{
    if (dep==m * n-k && x==x1 && y==y1 && dirs<=best){
        if(dirs<best) {best=dirs;count=1;save();}
        else count++;
        return;
    }
    if (dep==m * n-k || x==x1 && y==y1 || dirs>best) return;
    else
        for (int i=1; i<=8; i++)
            if (stepok(x+dx[i],y+dy[i])) {
                board[x+dx[i]][y+dy[i]]=dep+1;
                if(di !=i) dirs++;
                search(dep+1,x+dx[i],y+dy[i],i);
                if(di !=i) dirs--;
                board[x+dx[i]][y+dy[i]]=0;
            }
}
```

stepok用于判断是否越界。

```
static boolean stepok(int x,int y)
{
    return (x>0 && x<=n && y>0 && y<=m && board[x][y]==0);
}
```

save保存找到的解。

```
static void save()
{
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            bestb[i][j]=board[i][j];
}
```

对于当前位置还可加入剪枝函数live提早判断无解，进行剪枝。

```
static boolean live(int x,int y,int dep)
{
    int nm=n*m;
    if(d[x][y]>1 && endpoint>1) return false;
    for(int j=1;j<=8;j++){
        int p=x+dx[j],q=y+dy[j];
        if(stepok(p,q) && d[p][q]<2 && dep<nm-k-2) return false;
    }
    return true;
}
```

加入剪枝函数后的回溯法如下：

```
static void search (int dep,int x,int y,int di)
{
    if (dep==m*n-k && x==x1 && y==y1 && dirs<=best){
        if(dirs<best) {best=dirs;count=1;save();}
        else count++;
        return;
    }
    if (dep==m*n-k || x==x1 && y==y1 || dirs>best) return;
    else
        for (int i=1; i<=8; i++){
            int p=x+dx[i],q=y+dy[i];
            if (stepok(p,q) && live(p,q,dep)) {
                save(p,q,dep);
                if(di !=i) dirs++;
                search(dep+1,p,q,i);
                if(di !=i) dirs--;
                restore(p,q);
            }
        }
}
```

算法实现题 5-13　工作分配问题

★ 问题描述

设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法，为每一个人都分配 1 件不同的工作，并使总费用达到最小。

★ 算法设计

设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（其中 $1\leqslant n\leqslant 20$）。接下来的 n 行，每行 n 个数，表示工作费用。

★ 结果输出

将计算出的最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	9
10 2 3	
2 3 4	
3 4 5	

分析与解答：

此题的解空间显然是一棵排列树，可以套用搜索排列树的回溯法框架。

```
static void backtrack(int t)
{
    if (t>n) compute();
    else
        for (int j=t; j<=n; j++) {
            MyMath.swap(r,t,j);
            backtrack(t+1);
            MyMath.swap(r,t,j);
        }
}
```

其中，compute 计算当前方案的费用。

```
static void compute()
{
    int temp=0;
    for (int i=1;i<=n;i++) temp+=p[i][r[i]];
    if (temp<best){
        best=temp;
        for(int i=1;i<=n;i++) bestr[i]=r[i];
    }
}
```

算法实现题 5-14　独立钻石跳棋问题

★ 问题描述

独立钻石跳棋(如图 5-7 所示)的棋盘上有 33 个方格，每个方格中可放 1 枚棋子。棋盘中最多可摆放 32 枚棋子。下棋的规则是任一棋子可以沿水平或垂直方向跳过与其相邻的棋子进入空着的方格并吃掉被跳过的棋子。试设计一个算法，对于任意给定的棋盘布局，找出一种下棋方法，使得最终棋盘上只剩下一个棋子。

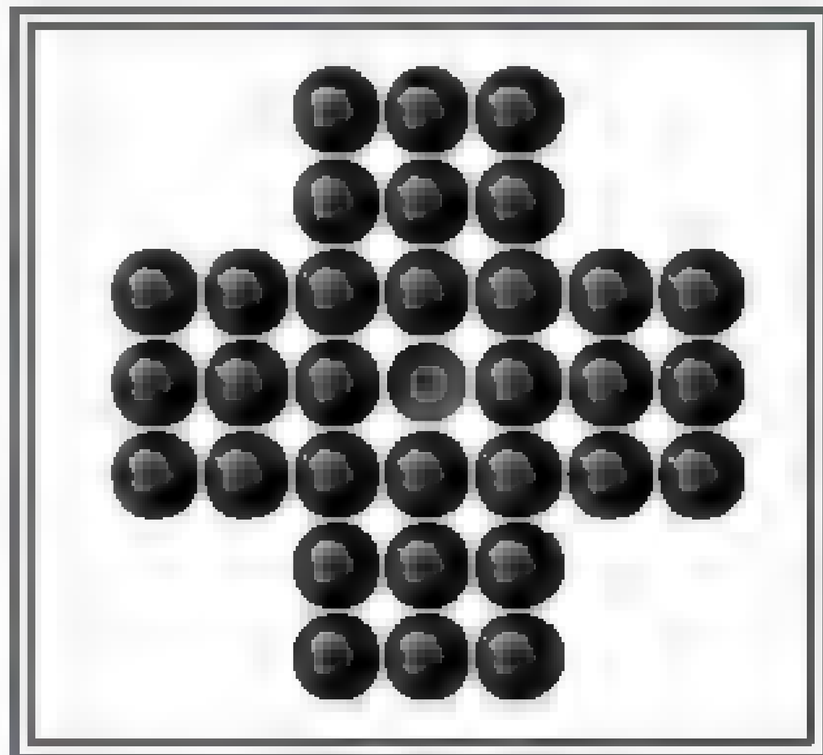


图 5-7　独立钻石跳棋

★ 算法设计

对于给定的独立钻石跳棋的棋盘初始布局，和棋盘上最终剩下的棋子所在的位置(x,y)，计算一种遵循下棋的规则下棋方法，使最终棋盘上仅在位置(x,y)处有 1 枚棋子。当$(x,y)=(0,0)$时，表示不指定棋子的最终位置。棋子位置的

坐标定义如图 5 8 所示。

		(2,0)	(3,0)	(4,0)		
		(2,1)	(3,1)	(4,1)		
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
		(2,5)	(3,5)	(4,5)		
		(2,6)	(3,6)	(4,6)		

图 5-8 棋子位置的坐标

★ **数据输入**

由文件 input.txt 提供输入数据。文件的第 1 行中有 1 个正整数 n，表示棋盘的初始布局中有 n 个棋子。第 2 行起每行 2 个数，分别是 n 个棋子的位置。最后 1 行是棋盘上最终剩下的棋子所在的位置。图 5-9 是棋盘的初始布局。

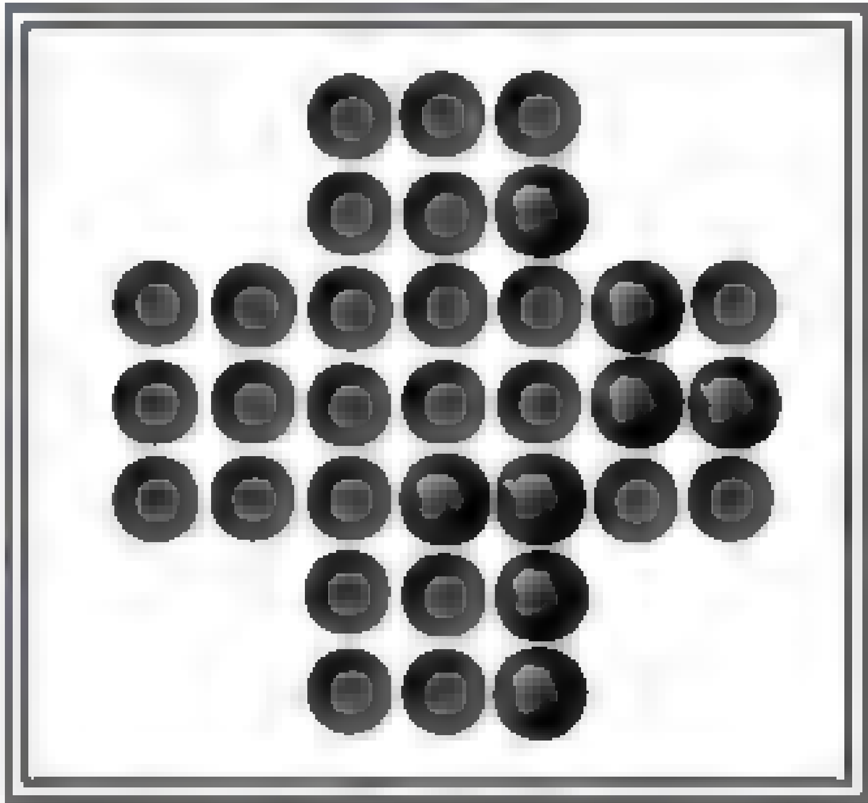


图 5-9 棋盘的初始布局

★ **结果输出**

将计算出的下棋步法依次输出到文件 output.txt 中。每行有 2 对方格坐标(a,b)和(c,d)，表示从方格(a,b)跳到方格(c,d)。问题无解时，则输出“No solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
8	(3,4) (5,4)
4 1	(4,6) (4,4)
5 2	(4,4) (6,4)
5 3	(6,4) (6,2)
6 3	(6,2) (4,2)
3 4	(4,1) (4,3)
4 4	(5,3) (3,3)
4 5	
0 0	

分析与解答：

1) 算法思想

解独立钻石跳棋问题的回溯算法框架如下：

```
static boolean backtrack1(int t)
{
    if(finish(t)) return true;
```

```
    for each step m{
        if(ok(m)){
            next(m);
            if(backtrack(t+1)) return true;
            restore(m);
        }
        return false;
    }
}
```

2）基本数据结构

用 board[7][7]表示棋盘。board[x][y]=0，表示空方格；board[x][y]=1，表示棋子占用方格；board[x][y]=2，表示棋盘外方格。初始棋盘为空棋盘，根据输入数据填入棋子。

```
static void init()
{
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++)
            if((i<2 || i>4) && (j<2 || j>4)) board[i][j]=2;
            else board[i][j]=0;
}
```

用类 Move 表示棋子的走步。

```
public class Move {int sx,sy,xv,yv;}
```

其中，(sx,sy)是起步坐标，(xv,yv)是走步方向。

搜索过程将产生许多棋盘状态。如何有效地存储这些棋盘状态。棋盘共有 33 个位置。如果用 1 位表示 1 个棋盘位置的状态，则一个棋盘需用 33 位来表示。每一位与棋盘位置的对应关系如图 5-10 所示。

		(2,0) 1	(3,0) 2	(4,0) 3		
		(2,1) 4	(3,1) 5	(4,1) 6		
(0,2) 7	(1,2) 8	(2,2) 9	(3,2) 10	(4,2) 11	(5,2) 12	(6,2) 13
(0,3) 14	(1,3) 15	(2,3) 16	(3,3) 17	(4,3) 18	(5,3) 19	(6,3) 20
(0,4) 21	(1,4) 22	(2,4) 23	(3,4) 24	(4,4) 25	(5,4) 26	(6,4) 27
		(2,5) 28	(3,5) 29	(4,5) 30		
		(2,6) 31	(3,6) 32	(4,6) 33		

图 5-10　用 33 位来表示棋盘

函数 coord2bit 将棋盘坐标转换为相应的位。

```
static long coord2bit(int x,int y)
{
```

```
    int pos;
    switch(y){
        case 0:     pos=x-2;break;
        case 1:     pos=x+1;break;
        case 5:     pos=x+25;break;
        case 6:     pos=x+28;break;
        default:    pos=x-8+y*7;
    }
    return ((long)1)<<pos;
}
```

算法的难点在于如何对当前棋局产生所有合法走步。一个可行的方法是,一次性产生所有可能的走步,存储在一个表中,算法通过对表的扫描来产生下一个合法走步。这个任务由算法 caches 和 cache 完成。

```
static void caches()
{
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++)
        {
            if(i<6 && i>0){cache(i,j,1,0);cache(i,j,-1,0);}
            if(j<6 && j>0){cache(i,j,0,1);cache(i,j,0,-1);}
        }
}

static void cache(int x, int y, int xv, int yv)
{
    if((board[x][y]!=2) && (board[x+xv][y+yv]!=2) && (board[x-xv][y-yv]!=2))
    {
        Move m=new Move();
        m.sx=x-xv;m.sy=y-yv;
        m.xv=xv;m.yv=yv;
        moves[count++]=m;
    }
}
```

所有 76 个合法走步存储在数组 moves 中。

二维数组 gmoves 存储相应的走步状态,用于判定走步的合法性。算法根据数组 moves 产生 gmoves 的值。

```
static void genmv()
{
    for(int j=0;j<count;j++){
        Move m=moves[j];
        gmoves[j][0]=coord2bit(m.sx,m.sy);
        gmoves[j][0]=coord2bit(m.sx+m.xv,m.sy+m.yv);
        gmoves[j][0]=coord2bit(m.sx+2*m.xv,m.sy+2*m.yv);
        gmoves[j][1]=coord2bit(m.sx,m.sy)|coord2bit(m.sx+m.xv,m.sy+m.yv);
```

```
}
```

算法中用一个全局变量 gboard 表示当前棋盘状态，由算法 initreg 对其进行初始化。

```
static void initreg()
{
    gboard=0;
    for(int i=0;i<7;i++)
        for(int j=0;j<7;j++)
            if(board[i][j]==1)
                gboard|=coord2bit(i,j);
}
```

函数 ok 判定走步 move 的合法性。

```
static boolean ok(long []move)
{
    return ((gboard & move[0])==move[1]);
}
```

函数 next 将棋盘状态变换为走步 move 后的状态。

```
static void next(long []move)
{
    gboard=move[0];
}
```

函数 restore 将棋盘状态恢复为走步 move 前的状态。

```
static void restore(long []move)
{
    gboard=move[0];
}
```

在算法搜索过程中，不同的搜索路径可到达同一个棋盘状态。为了避免无效搜索，可将搜索过的棋盘状态存储起来。用 makesp 申请内存块。

```
static void makesp(int mm)
{
    int i,cnt=mm/blksize+1;
    sptab=new byte[mm];
    for(i=0;i<cnt;i++)sptab[i]=0;
}
```

save 将当前棋盘状态 gboard 存储到内存表 sptab 中。

```
static void save()
{
    sptab[(int)(gboard/8)]|=(1<<(gboard&0x07));
```

```
}
```

tried 检测当前棋盘状态 gboard 是否已存储在内存表中。

```
static boolean tried()
{
    return ((long)sptab[(int)(gboard/8)]&(1<<(gboard&0x07)))>0;
}
```

3) 算法实现

在上述讨论的基础上,解独立钻石跳棋问题的回溯法可具体描述如下:

```
static boolean backtrack(int t)
{
    long []m;
    if(finish(t)) return true;
    if(t<26 && t%2==0 && tried())return false;
    for(int i=0;i<count;i++){
        m=gmoves[i];
        if(ok(m)){
            next(m);
            if(backtrack(t+1)){
                ans[t]=find(m);
                return true;
            }
            restore(m);
        }
    }
    save();
    return false;
}
```

其中,finish 检测算法终止条件。(endx,endy)是终止方格的坐标;num 是初始棋盘上棋子的个数。

```
static boolean finish(int t)
{
    if (endx>0 || endy>0) return gboard==coord2bit(endx,endy);
    else return t>num-2;
}
```

find 找出与走步状态相应的走步描述。

```
static Move find(long []m)
{
    for(int i=0; i<count; i++)
        if(gmoves[i]==m)return moves[i];
    return null;
}
```

实现算法的主函数描述如下：

```
public static void main(String [ ] args)
{
    ReadStream keyboard=new ReadStream();
    num=keyboard.readInt();
    init();
    for(int i=0;i<num;i++){
        int x=keyboard.readInt();
        int y=keyboard.readInt();
        board[x][y]=1;
    }
    endx=keyboard.readInt();
    endy=keyboard.readInt();
    caches();
    genmv();
    initreg();
    makesp(((int) 1)<<30);
    if(backtrack(0))for(int i=0;i<num-1;i++)out(ans[i]);
    else System.out.println("No solution!");
}
```

函数 out 输出走步。

```
static void out(Move m)
{
    System.out.println("("+m.sx+","+m.sy+") ("+(m.sx+2*m.xv)+","+(m.sy+
    2*m.yv)+")");
}
```

算法实现题 5-15　智力拼图问题

★ **问题描述**

设有 12 个平面图形如图 5-11 所示。每个图形的形状互不相同，但它们都是由 5 个大小相同的正方形组成。图 5-11 中 12 个图形拼接成一个 6×10 的矩形。试设计一个算法，计算出用这 12 个图形拼接成给定矩形的拼接方案。

★ **算法设计**

对于给定矩形，计算用上述 12 个图形拼接成给定矩形的一个拼接方案。拼接方案中每个图形可以经过旋转或翻转后进行拼接，但要求使用 12 个图形中每个图形恰好 1 次。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n，表示给定的矩形是一个 $m\times n$ 矩形，且 $m\times n=60$。

★ **结果输出**

将计算出的矩形的拼接方案输出到文件 output.txt。每行 n 个字符，共 m 行。

给定的 12 个图形的编号如图 5-12 所示。如果不存在所要求的拼接方案，则输出“No solution!”。

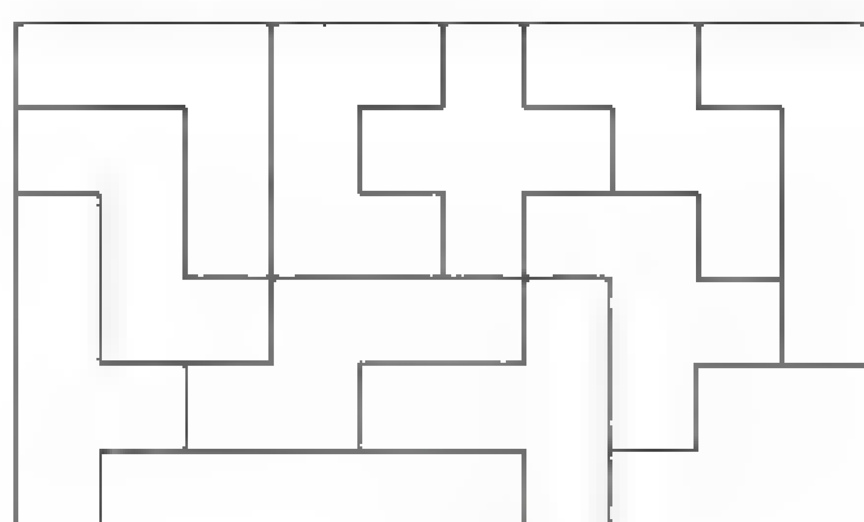


图 5-11　智力拼图

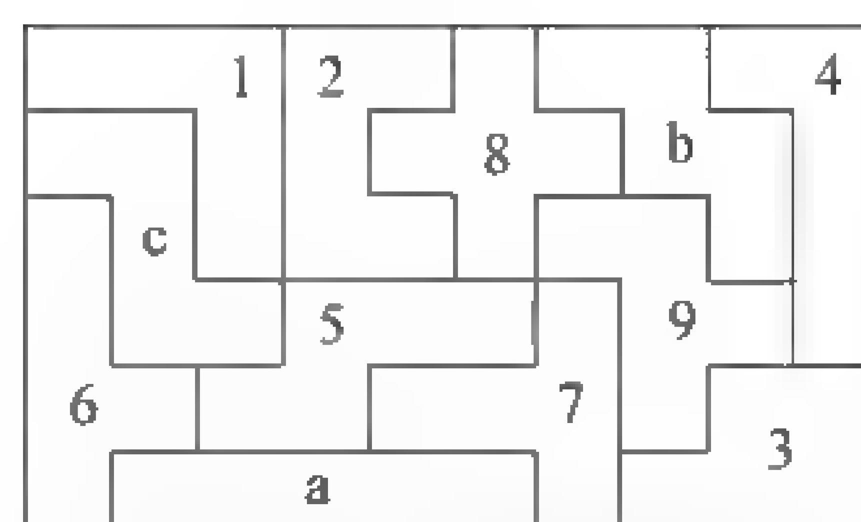


图 5-12　智力拼图图形编号

输入文件示例	输出文件示例
input. txt	output. txt
6 10	111c9aaaaa
	1ccc999777
	1c3339bb74
	22233bb874
	26255b8884
	6666555844

分析与解答：

1）算法思想

将矩形划分成 $m\times n$ 个方格。按从上到下和从左到右的次序，考查每一个未覆盖方格。每个方格都可能被12个图形中的一个图形所覆盖。每一个图形经过旋转或翻转，最多可能产生8个不同形态。对每种可能进行无遗漏的回溯搜索，直至找到一个解。

2）基本数据结构

全局变量 brow 和 bcol 分别表示 m 和 n 的值。

用数组 aa[5][5]表示给定的每个图形。定义类 dom 如下：

```
public class dom{
    public byte [][]aa=new byte[5][5];
    public dom() {
        for(int i=0;i<5;i++)
            for(int j=0;j<5;j++) aa[i][j]=0;
    }
    public dom(byte b[][]) {
        for(int i=0;i<5;i++)
            for(int j=0;j<5;j++) aa[i][j]=b[i][j];
    }
}
```

originit 将每个图形结构按其编号存储于数组 orig 中。

```
static void originit()
{
    byte [ ][ ]pc=new byte[5][5];
    pcinit(pc);
```

```
pc[0][0]=1;pc[0][1]=1;pc[0][2]=1;
pc[1][0]=1;
pc[2][0]=1;
orig[0]=new dom(pc);

pcinit(pc);
pc[0][0]=2;pc[0][1]=2;
pc[1][0]=2;
pc[2][0]=2;pc[2][1]=2;
orig[1]=new dom(pc);

pcinit(pc);
pc[0][0]=3;pc[0][1]=3;
pc[1][0]=3;pc[1][1]=3;
pc[2][0]=3;
orig[2]=new dom(pc);

pcinit(pc);
pc[0][0]=4;pc[0][1]=4;
pc[1][0]=4;
pc[2][0]=4;
pc[3][0]=4;
orig[3]=new dom(pc);

pcinit(pc);
pc[0][0]=5;
pc[1][0]=5;pc[1][1]=5;
pc[2][1]=5;
pc[3][1]=5;
orig[4]=new dom(pc);

pcinit(pc);
pc[0][0]=6;
pc[1][0]=6;pc[1][1]=6;
pc[2][0]=6;
pc[3][0]=6;
orig[5]=new dom(pc);

pcinit(pc);
pc[0][0]=7;pc[0][1]=7;pc[0][2]=7;
pc[1][1]=7;
pc[2][1]=7;
orig[6]=new dom(pc);

pcinit(pc);
pc[0][1]=8;
pc[1][0]=8;pc[1][1]=8;pc[1][2]=8;
pc[2][1]=8;
orig[7]=new dom(pc);

pcinit(pc);
```

```
        pc[0][1]=9;
        pc[1][0]=9;pc[1][1]=9;
        pc[2][1]=9;pc[2][2]=9;
        orig[8]=new dom(pc);

        pcinit(pc);
        pc[0][0]=10;pc[0][1]=10;pc[0][2]=10;pc[0][3]=10;pc[0][4]=10;
        orig[9]=new dom(pc);

        pcinit(pc);
        pc[0][1]=11;pc[0][2]=11;
        pc[1][0]=11;pc[1][1]=11;
        pc[2][0]=11;
        orig[10]=new dom(pc);

        pcinit(pc);
        pc[0][0]=12;pc[0][1]=12;
        pc[1][1]=12;
        pc[2][1]=12;pc[2][2]=12;
        orig[11]=new dom(pc);
    }
```

用类 fboard 表示给定矩形的状态。

```
public class fboard{
    byte []pos=new byte[60];
    byte []mark=new byte[12];
    public fboard() {
        for(int a=0;a<12;a++)mark[a]=0;
        for(int a=0;a<60;a++)pos[a]=0;
    }
    public fboard(fboard x) {
        for(int a=0;a<12;a++)mark[a]=x.mark[a];
        for(int a=0;a<60;a++)pos[a]=x.pos[a];
    }
    public void copy(fboard x) {
        for(int a=0;a<12;a++)mark[a]=x.mark[a];
        for(int a=0;a<60;a++)pos[a]=x.pos[a];
    }
}
```

其中,数组 mark 用于标记使用过的图形。pos 用于表示矩形中 60 个方格被覆盖的情况。在矩形方格(row,col)处的覆盖状态存储在 pos[k]中,数组下标 k 的值由 addr 计算如下:

```
static int addr(int row,int col)
{
    return (row  1) * bcol + col  1;
```

3）算法实现

算法一开始先作初始化工作，由 init 来完成。

```
static void init()
{
    originit();
    for (int b=0;b<12;b++){
        dom tmp1=new dom(orig[b].aa);
        for(int d=0;d<5;d++)
            for(int c=0;c<5;c++)
                if(tmp1.aa[d][c]>0)tmp1.aa[d][c]|=0x10;
        int vc=0;
        for(int a=0;a<8;a++){
            boolean exist=false;
            for(int c=0;c<vc && !exist;c++)
                if(domeq(var[b][c],tmp1))exist=true;
            if(!exist) var[b][vc++]=new dom(tmp1.aa);
            dom tmp2=new dom(tmp1.aa);
            rotp(tmp1,tmp2);      // 旋转
            if (a==3){
                tmp2=new dom(tmp1.aa);
                flip(tmp1,tmp2); // 翻转
            }
        }
        nvar[b]=vc;
    }
    nvar[0]=1;
}
```

其中，全局数组 var[12][8]用于存储 12 个图形中每个图形的不同形态，nvar[12]用于存储 12 个图形中每个图形的不同形态数。图形旋转和图形翻转分别由 rotp 和 flip 完成。

```
static void rotp(dom des, dom src)
{
    dominit(des);
    for(int x=4,flag=0,xp=0;x>=0;x--){
        for(int y=0;y<5;y++){
            des.aa[xp][y]=src.aa[y][x];
            if(src.aa[y][x]>0)flag=1;
        }
        if(flag>0)xp++;
    }
}

static void flip(dom des, dom src)
{
    dominit(des);
```

```
        for(int x=4,flag=0,xp=0;x>=0;x    ){
            for(int y=0;y<5;y++){
                des.aa[y][xp]=src.aa[y][x];
                if(src.aa[y][x]>0)flag=1;
            }
            if(flag>0)xp++;
        }
    }
```

算法的主体是对矩形覆盖的回溯搜索。

```
    static void search(fboard pre)
    {
        int npla=0,frow=0,fcol=0;
        fboard ff=new fboard (pre);
        if(ans>0)return;
        // 找最左上的未覆盖方格
        for(int row=1,found=0;row<=brow && found==0;row++)
            for(int col=1;col<=bcol && found==0;col++)
                if (pre.pos[addr(row,col)]==0){
                    frow=row;fcol=col;found=1;
                }
        // 计算已用过的图形数
        for (int pn=0;pn<12;pn++)
            if (pre.mark[pn]>0) npla++;
        for (int pn=0;pn<12;pn++){
            // 是否已用过
            if (pre.mark[pn]>0) continue;
            // 对每种不同形态
            for (int pv=0;pv<nvar[pn];pv++){
                dom tryp=var[pn][pv];
                int py=frow;
                for (int px=1;px<=bcol;px++){
                    if (px>fcol) break;
                    if(check(tryp,ff,px,py)){
                        // 用该图形覆盖
                        for (int row=0;row<5;row++)
                            for(int col=0;col<5;col++)
                                if(tryp.aa[row][col]>0)
                                   ff.pos[addr(row+py,col+px)]|=tryp.aa[row][col];
                        ff.mark[pn]=1;
                        if (npla>10){ ++ans;outf(ff);return;}
                        else search(ff);
                        // 回溯
                      ff.copy (pre);
                    }//if(ok)
```

```
            }//px
        }//pv
    }//pn
}
```

其中，算法check用于检测图形覆盖的可行性。outf输出找到的图形覆盖方案。

```
static boolean check(dom tryp,fboard ff,int px,int py)
{
    for (int row=0;row<5;row++)
        for(int col=0;col<5;col++)
            if (tryp.aa[row][col]>0){
                int r=row+py,c=col+px;
                if(r>brow || c>bcol || ff.pos[addr(r,c)]>0)return false;
            }
    return true;
}

static void outf(fboard ff)
{
    if(flg){
        for (int col=1;col<=bcol;col++){
            for (int row=1;row<=brow;row++){
                int x=ff.pos[addr(row,col)]-16;
                if(x<10)System.out.print(x);
                if(x==10)System.out.print("a");
                if(x==11)System.out.print("b");
                if(x==12)System.out.print("c");
            }
            System.out.println();
        }
    }
    else{
        for (int row=1;row<=brow;row++){
            for (int col=1;col<=bcol;col++){
                int x=ff.pos[addr(row,col)]-16;
                if(x<10)System.out.print(x);
                if(x==10)System.out.print("a");
                if(x==11)System.out.print("b");
                if(x==12)System.out.print("c");
            }
            System.out.println();
        }
    }
}
```

执行算法的主函数如下：

```
public static void main(String [ ] args)
{
    ReadStream keyboard=new ReadStream();
    brow=keyboard.readInt();
    bcol=keyboard.readInt();
    if(brow<bcol){int tmp=brow;brow=bcol;bcol=tmp;flg=true;}
    if(bcol<3 || brow * bcol!=60)System.out.println("No solution!");
    else{
        init();
        fboard ff=new fboard();
        search(ff);
    }
}
```

算法实现题 5-16　布线问题

★ 问题描述

假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i,j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r,s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\sum_{1 \leqslant i < j \leqslant n} \text{conn}(i,j)\text{dist}(r,s)$。试设计一个算法找出所给 n 个元件的布线成本最小的布线方案。

★ 算法设计

设计一个算法，对于给定的 n 个元件，计算最佳布线方案，使布线费用达到最小。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（其中 $1 \leqslant n \leqslant 20$）。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数，$1 \leqslant i < j \leqslant 20$。

★ 结果输出

将计算出的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

分析与解答：

与教材中的电路板排列问题类似。回溯法如下：

```
static void backtrack(int i)
{
    if (i==n) {
        int tmp=len(i);
        if(tmp<bestd){bestd=tmp;for (int j=1;j<=n;j++) bestx[j]=x[j];}
```

```
        }
        else
            for (int j=i; j<=n; j++) {
                MyMath.swap(x,i,j);
                int ld=len(i);
                if (ld<bestd) backtrack(i+1);
                MyMath.swap(x,i,j);
            }
    }
```

其中，len 计算布线费用。

```
    static int len(int ii)
    {
        int sum=0;
        for (int i=1; i<=ii; i++)
            for(int j=i+1;j<=ii;j++){
                int dist=x[i]>x[j]? x[i]-x[j]:x[j]-x[i];
                sum+=conn[i][j] * dist;
            }
        return sum;
    }
```

算法实现题 5-17　最佳调度问题

★ 问题描述

假设有 n 个任务由 k 个可并行工作的机器完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

★ 算法设计

对任意给定的整数 n 和 k，以及完成任务 i 需要的时间为 t_i，$i=1\sim n$。计算完成这 n 个任务的最佳调度。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。第 2 行的 n 个正整数是完成 n 个任务需要的时间。

★ 结果输出

将计算出的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	17
2 14 4 16 6 5 3	

分析与解答：

简单回溯搜索。

```
    static void search(int dep)
    {
```

```
    if(dep==n) {
        int tmp=comp();
        if(tmp<best) best=tmp;
        return;
    }
    for(int i=0;i<k;i++){
        len[i]+=t[dep];
        if(len[i]<best) search(dep+1);
        len[i]-=t[dep];
    }
}
```

comp 计算完成任务的时间。

```
static int comp()
{
    int tmp=0;
    for(int i=0;i<k;i++) if(len[i]>tmp)tmp=len[i];
    return tmp;
}
```

算法实现题 5-18　无优先级运算问题

★ 问题描述

给定 n 个正整数和 4 个运算符+、-、*、/，且运算符无优先级，如 2+3*5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次，但每种运算符可以任意使用。

★ 算法设计

对于给定的 n 个正整数，设计一个算法，用最少的无优先级运算次数产生整数 m。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

★ 结果输出

将计算出的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

分析与解答：

readin 读入初始数据。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
```

```
        n=keyboard.readInt();
        m=keyboard.readInt();
        a=new int[n];
        num=new int[n];
        oper=new int[n];
        flag=new int[n];
        for(int i=0;i<n;i++) {a[i]=keyboard.readInt();flag[i]=0;}
}
```

采用迭代加深的回溯法。

```
static boolean search(int dep)
{
    if(dep>k) {if(found()) return true; else return false;}
    for(int i=0;i<n;i++)
        if (flag[i]==0){
            num[dep]=a[i];
            flag[i]=1;
            for(int j=0;j<4;j++){
                oper[dep]=j;
                if(search(dep+1)) return true;
            }
            flag[i]=0;
        }
    return false;
}
```

found 判断是否找到解。

```
static boolean found()
{
    int x=num[0];
    for(int i=0;i<k;i++){
        switch (oper[i]){
            case 0: x+=num[i+1]; break;
            case 1: x-=num[i+1]; break;
            case 2: x*=num[i+1]; break;
            case 3: x/=num[i+1]; break;
        }
    }
    return (x==m);
}
```

实现算法的主函数如下：

```
public static void main(String[] args)
{
    readin();
```

```
        for(k=0;k<n;k++)
            if(search(0)) {System.out.println(k);out();return;}
        System.out.println("No Solution!");
    }
```

算法实现题 5-19　世界名画陈列馆问题

★ 问题描述

世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗,需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外,还可以监视与它所在的陈列室相邻的上、下、左、右4个陈列室。试设计一个安排警卫机器人哨位的算法,使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下,且所用的警卫机器人数最少。

★ 算法设计

设计一个算法,计算警卫机器人的最佳哨位安排,使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下,且所用的警卫机器人数最少。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 m 和 n(其中 $1 \leqslant m, n \leqslant 20$)。

★ 结果输出

将计算出的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第1行是警卫机器人数;接下来的 m 行中每行 n 个数,0表示无哨位,1表示哨位。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

分析与解答:

1) 状态空间树

本题的状态空间树是一棵子集树,可用对子集树进行搜索的回溯算法框架求解。依从上到下、从左到右的顺序依次考查每一个陈列室设置警卫机器人哨位的情况,以及该陈列室受警卫机器人监视的情况。用 $x[i,j]$ 表示陈列室 (i,j) 当前设置警卫机器人哨位的状态。当 $x[i,j]=1$ 时,表示已经在陈列室 (i,j) 设置了警卫机器人哨位;当 $x[i,j]=0$ 时,表示在陈列室 (i,j) 尚未设置警卫机器人哨位。另外,用 $y[i,j]$ 表示陈列室 (i,j) 当前受警卫机器人监视的状态。当 $y[i,j]=1$ 时,表示陈列室 (i,j) 已受警卫机器人监视;当 $y[i,j]=0$ 时,表示在陈列室 (i,j) 尚未受警卫机器人监视。

2) 采用剪枝技术提高回溯法的效率

(1) 下界剪枝法。

设当前已设置的警卫机器人哨位数为 k,已受警卫机器人监视的陈列室数为 t,当前最优警卫机器人哨位数为best。在一般情况下,可根据 k 和 t 的值,估计出尚需设置的警卫机

器人哨位数下界 $f(k,t)$。当 $f(k,t) \geq$ best 时，可将状态空间树中以当前结点为根的子树剪去。

(2) 控制剪枝法。

设 p 和 q 是状态空间树中两个不同的结点。如果按照结点 p 和 q 的某一相关关系，可以确定以结点 q 为根的子树中的解不优于以结点 p 为根的子树中的解，则称结点 p 控制了结点 q。对于本题来说，可考虑以下的结点控制关系。

① 已受监视结点的控制关系。

设在回溯搜索时当前所关注的是陈列室(i,j)。该陈列室已受监视，即 $y[i,j]=1$。与其相邻的其他陈列室的受监视状态如图 5-13 所示。

此时在陈列室(i,j)处设置一个警卫机器人哨位，即取 $x[i,j]=1$，相应于状态空间树中的一个结点 q。在陈列室$(i+1,j+1)$处设置一个警卫机器人哨位，即取 $x[i+1,j+1]=1$，相应于状态空间树中的另一个结点 p。容易看出此时以结点 q 为根的子树中的解不优于以结点 p 为根的子树中的解，即结点 p 控制了结点 q。可将状态空间树中以结点 q 为根的子树剪去。由此总结出在以从上到下、从左到右的顺序依次考查每一个陈列室时，已受监视的陈列室处不必设置警卫机器人哨位。这样可以避免许多无效搜索。

② 测试结点的控制关系

设陈列室(i,j)是当前以从上到下、从左到右的顺序搜索遇到的第 1 个未受监视的陈列室。为了使陈列室(i,j)受到监视，可在陈列室$(i+1,j)$、(i,j)、$(i,j+1)$处设置警卫机器人哨位。在这 3 处设置哨位所相应的状态空间树中的结点分别为 p、q 和 r，如图 5-14 所示。

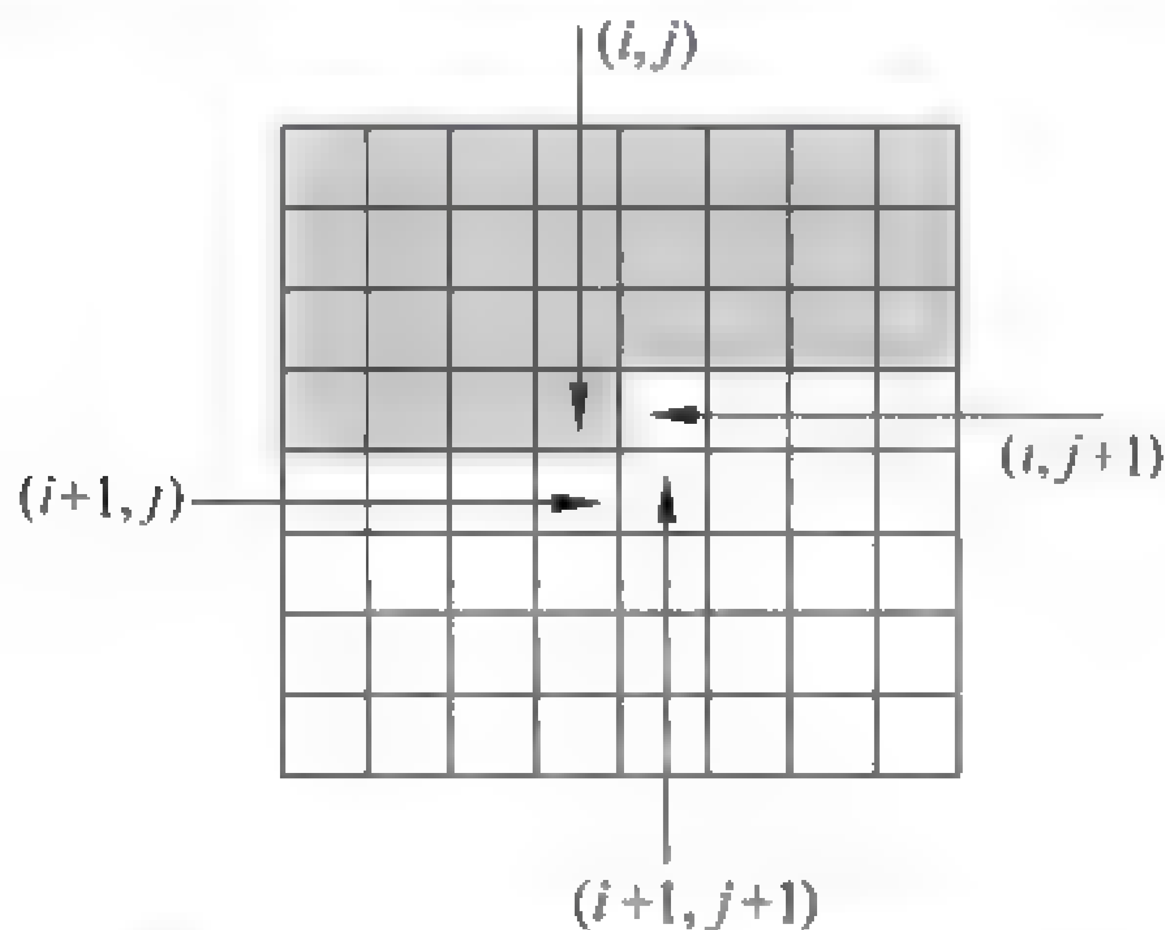


图 5-13　受监视结点的控制关系

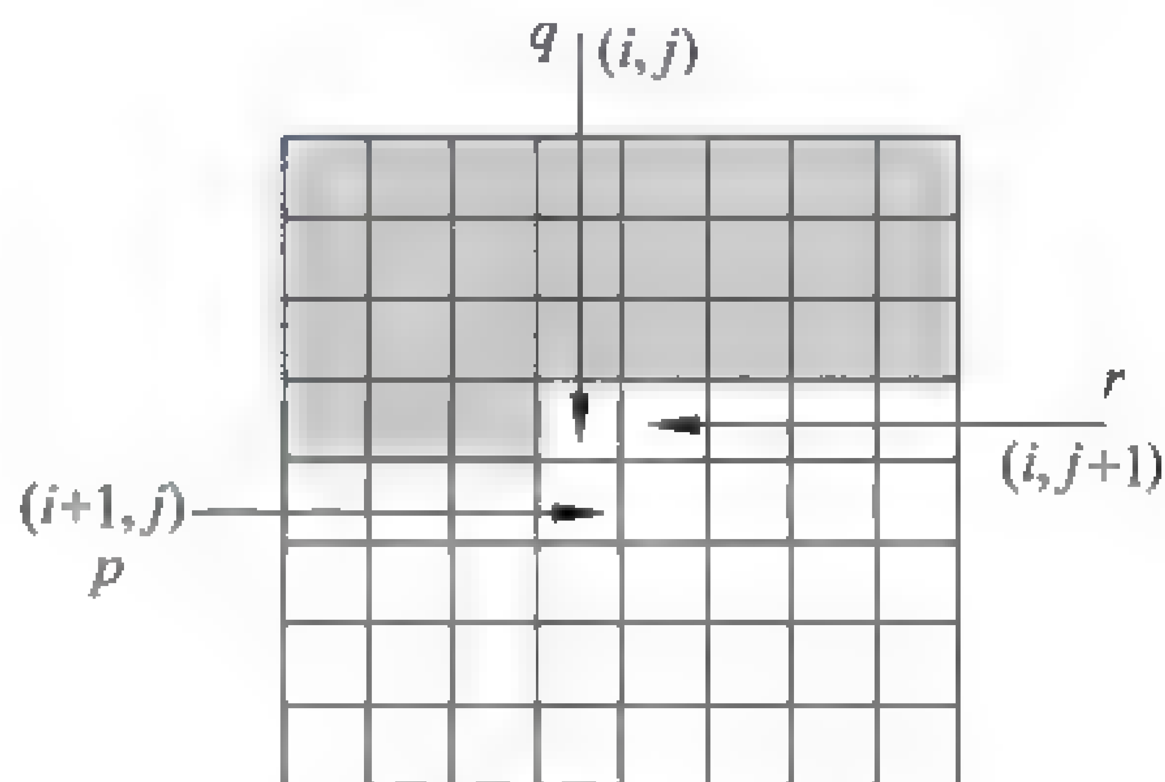


图 5-14　测试结点的控制关系

显而易见，当 $y(i,j+1)=1$ 时，结点 p 控制结点 q；当 $y(i,j+1)=1$ 且 $y(i,j+2)=1$ 时，结点 p 控制结点 r。因此，在搜索时应按 $p \to q \to r$ 的顺序来扩展结点，并检测结点 p 对结点 q 和结点 r 的控制条件，及时剪去受控结点所相应的子树。

3) 简化边界条件

在陈列室矩形阵列的外圈扩展一层，即增加 0 行和 0 列，n+1 行和 m+1 列。使算法可以统一地处理边界条件。

根据前面的分析，可以设计安排警卫机器人哨位的回溯法如下。

算法中用到的变量类型说明如下：

```
static final int MLEN=50;
```

```
static int n,m,best,k=0,t=0,t1,t2,more;
static boolean p;
static int [][]d={{0,0,0},{0,0,0},{0,0,-1},{0,-1,0},{0,0,1},{0,1,0}};
static int [][]x=new int[MLEN+1][MLEN+1];
static int [][]y=new int[MLEN+1][MLEN+1];
static int [][]bestx=new int[MLEN+1][MLEN+1];
```

回溯法的主体由 search(i,j)来实现。参数 i 和 j 表示当前搜索位置。

```
static void search(int i,int j)
{
    do{
        j++;
        if(j>m){i++;j=1;}
    }while (!((y[i][j]==0) || (i>n)));
    if(i>n){
        if(k<best){best=k;copy(bestx,x);}
        return;
    }
    if(k+(t1-t)/5>=best) return;
    if((i<n-1) && (k+(t2-t)/5>=best)) return;
    if((i<n)){
        change(i+1,j);
        search(i,j);
        restore(i+1,j);
    }
    if((j<m)&&((y[i][j+1]==0) || (y[i][j+2]==0))){
        change(i,j+1);
        search(i,j);
        restore(i,j+1);
    }
    if(((y[i+1][j]==0)&&(y[i][j+1]==0))){
        change(i,j);
        search(i,j);
        restore(i,j);
    }
}
```

上述算法中，change(i,j)用于在(i,j)处设置一个警卫机器人哨位，并相应地改变其相邻陈列室的受监视状况。

```
static void change(int i,int j)
{
    x[i][j]=1;k++;
    for (int s=1;s<=5;s++) {
        int p=i+d[s][1];
        int q=j+d[s][2];
```

```
            y[p][q]++;
            if((y[p][q]==1)) t++;
        }
}
```

restore(i,j)则用于撤销在(i,j)处设置的警卫机器人哨位，并相应地改变其相邻陈列室的受监视状况。

```
static void restore(int i,int j)
{
    x[i][j]=0;k--;
    for (int s=1;s<=5;s++){
        int p=i+d[s][1];
        int q=j+d[s][2];
        y[p][q]--;
        if((y[p][q]==0))t--;
    }
}
```

最后由 compute 调用主体算法，搜索最优解。

```
static void compute()
{
    more=m/4+1;
    if(m%4==3)more++;
    else if(m%4==2) more+=2;
    t2=m*n+more+4;
    t1=m*n+4;
    best=Integer.MAX_VALUE;
    for(int i=0;i<=MLEN;i++)
        for(int j=0;j<=MLEN;j++){x[i][j]=y[i][j]=0;}
    if((n==1) && (m==1)) {
        System.out.println(1);System.out.println(1);
        return;
    }
    for (int i=0;i<=m+1;i++) {y[0][i]=1;y[n+1][i]=1;}
    for (int i=0;i<=n+1;i++) {y[i][0]=1;y[i][m+1]=1;}
    search(1,0);
    output();
}
```

算法实现题 5-20　世界名画陈列馆问题(不重复监视)

★ 问题描述

世界名画陈列馆由 $m\times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，并且要求每一个陈列室

仅受一个警卫机器人监视,且所用的警卫机器人数最少。

★ **算法设计**

设计一个算法,计算警卫机器人的最佳哨位安排,使得名画陈列馆中每一个陈列室都仅受一个警卫机器人监视,且所用的警卫机器人数最少。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有2个正整数 m 和 n(其中,$1 \leqslant m, n < 20$)。

★ **结果输出**

将计算出的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第1行是警卫机器人数;接下来的 m 行中每行 n 个数,0 表示无哨位,1 表示哨位。如果不存在满足要求的哨位安排,则输出"No Solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

分析与解答:

本题要求每一个陈列室仅受一个警卫机器人监视,在许多情况下问题无解。下面分3种情形来讨论。

1) $1=n \leqslant m$ 的情形

此时问题恒有解,且容易直接写出其最优解。

当 $m \bmod 3=1$ 时,将机器人哨位置于$(1,3k+1)$,$k=0,1,\cdots,\lfloor m/3 \rfloor$。

当 $m \bmod 3=0$ 或 2 时,将机器人哨位置于$(1,3k+2)$,$k=0,1,\cdots,\lfloor m/3 \rfloor$。

2) $2=n \leqslant m$ 的情形

在这种情况下,如果问题有解,则易知必须在两端分别设置2个机器人哨位。这2个机器人哨位各监视3个陈列室。其余的 k 个机器人哨位均监视4个陈列室。由此可见,$2m=4k+6$,即 $m=2k+3$ 为奇数。当 m 为偶数时问题无解。

当 m 为奇数时,容易直接写出其最优解。将机器人哨位分别置于$(1,4k+3)$和$(2,4k+1)$,$k=0,1,\cdots,\lfloor m/4 \rfloor$。

3) $2<n \leqslant m$ 的情形

当 $n>2$ 时,用直接枚举法容易验证 $n=3,m=3$ 和 $n=3,m=4$ 时问题无解;$n=4,m=4$ 时问题有解。当 $n \geqslant 3$ 且 $m \geqslant 5$ 时,问题无解。这一结论可证明如下。考虑左上角的 3×5 阵列。下面证明在不重复监视的前提下,无法使这 3×5 阵列中的每一个陈列室都受到监视。为此,考虑本问题的一个变形问题 $P(n,m)$如下。在设置警卫机器人的哨位时,允许在第 $n+1$ 行和第 $m+1$ 列设置哨位,但不要求第 $n+1$ 行和第 $m+1$ 列的陈列室均受监视。如果 $n \geqslant 3$ 且 $m \geqslant 5$ 时,在不重复监视的前提下原问题有解,则 $P(3,5)$一定有解。换句话说,如果问题 $P(3,5)$无解,则当 $n \geqslant 3$ 且 $m \geqslant 5$ 时,不重复监视问题无解。因此,问题转化为证明问题 $P(3,5)$无解。对算法实现题5-19解答的算法 search 进行适当修改,可用

于问题$P(n,m)$求解。用修改过的算法 search 对问题 $P(3,5)$求解得知该问题无解。这就证明了上述结论。

具体算法由 compute 实现如下：

```
static void compute()
{
    for(int i=0;i<=MAXLENGTH;i++)
        for(int j=0;j<=MAXLENGTH;j++)x[i][j]=0;
    boolean ok=false;
    if(n==1){
        int k=m/3;
        if(m%3==1) for(int j=0;j<=k;j++)x[1][3*j+1]=1;
        else {
            if(m%3==0)k--;
            for(int j=0;j<=k;j++)x[1][3*j+2]=1;
        }
        best=k+1;ok=true;
    }
    if(m==1){
        int k=n/3;
        if(n%3==1) for(int j=0;j<=k;j++)x[3*j+1][1]=1;
        else {
            if(n%3==0)k--;
            for(int j=0;j<=k;j++)x[3*j+2][1]=1;
        }
        best=k+1;ok=true;
    }
    if(n==2 && m%2==0){
        int k=m/4;
        if(m%4==0)k--;
        for(int j=0;j<=k;j++){x[1][4*j+3]=1;x[2][4*j+1]=1;}
        best=2*k+2;ok=true;
    }
    if(m==2 && n%2==0){
        int k=n/4;
        if(n%4==0)k--;
        for(int j=0;j<=k;j++){x[4*j+3][1]=1;x[4*j+1][2]=1;}
        best=2*k+2;ok=true;
    }
    if(n==4 && m==4){x[1][1]=1;x[1][4]=1;x[4][1]=1;x[4][4]=1;best=4;ok=true;}
    if(ok) output();
    else System.out.println("No Solution!");
}
```

算法实现题 5-21　部落卫队问题

★ 问题描述

原始部落 byteland 中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都

有他的仇敌。部落酋长为了组织一支保卫部落的队伍,希望从部落的居民中选出最多的居民入伍,并保证队伍中任何两个人都不是仇敌。

★ 算法设计

给定 byteland 部落中居民间的仇敌关系,计算组成部落卫队的最佳方案。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 m,表示 byteland 部落中有 n 个居民,居民间有 m 个仇敌关系。居民编号为 $1,2,\cdots,n$。接下来的 m 行中,每行有2个正整数 u 和 v,表示居民 u 与居民 v 是仇敌。

★ 结果输出

将计算出的部落卫队的最佳组建方案输出到文件 output.txt 中。文件的第1行是部落卫队的顶人数;文件的第2行是卫队组成 x_i,$1\leqslant i\leqslant n$,$x_i=0$ 表示居民 i 不在卫队中,$x_i=1$ 表示居民 i 在卫队中。

输入文件示例	输出文件示例
input.txt	output.txt
7 10	3
1 2	1 0 1 0 0 0 1
1 4	
2 4	
2 3	
2 5	
2 6	
3 5	
3 6	
4 5	
5 6	

分析与解答:

本题设计解最大独立集问题的回溯法。与主教材中最大团问题的解法十分相似。

```
static void backtrack(int i)
{
    if(i>n){
        for(int j=1;j<=n;j++) bestx[j]=x[j];
        bestn=cn;
        return;
    }
    boolean ok=true;
    for(int j=1;j<i;j++)
        if (x[j]>0 && a[i][j]){ok=false;break;}
    if(ok){
        x[i]=1;cn++;
        backtrack(i+1);
```

```
            x[i]=0;cn    ;
        }
        if(cn+n-i>bestn){x[i]=0;backtrack(i+1);}
    }
```

readin 读入初始数据。

```
    static void readin()
    {
        ReadStream keyboard=new ReadStream();
        n=keyboard.readInt();
        int e=keyboard.readInt();
        a=new boolean[n+1][n+1];
        for(int i=0;i<=n;i++)
            for(int j=0;j<=n;j++)a[i][j]=false;
        for(int i=1;i<=e;i++){
            int u=keyboard.readInt();
            int v=keyboard.readInt();
            a[u][v]=a[v][u]=true;
        }
    }
```

maxInde 作初始化,并用回溯法求解。

```
    static int maxInde(int []v)
    {
        x=new int[n+1];
        for(int i=0;i<=n;i++)x[i]=0;
        cn=0;bestn=0;bestx=v;
        backtrack(1);
        return bestn;
    }
```

算法实现题 5-22　虫蚀算式问题

★ 问题描述

虫蚀算式是指古书中算式的一部分被虫蛀了。虫蚀算式问题是根据虫蚀算式剩下的数字,逻辑推断被虫蛀了的数字。例如:

```
   4 3 ? 9 8 6 5 0 ? 4 5
+      8 4 6 8 ? 6 6 3 3
   4 4 4 4 5 5 0 6 9 7 8
```

其中,"?"表示虫蛀的数字。根据此虫蚀算式,容易推断出,第 1 行的 2 个虫蛀数字分别是 5 和 3,第 2 行的虫蛀数字是 5。

一般情况下,虫蚀算式问题假设,算式中所有数字都被虫蛀了,但是知道虫蚀算式中哪些数字相同。另外还知道虫蚀算式是 n 进制加法算式。虫蚀算式中的 3 个数都是 n 位数,且允许前导 0。

★ 算法设计

对于给定的虫蚀算式,计算算式中的虫蚀数字。

★ **数据输入**

由文件 input.txt 给出输入数据。文件有4行。第1行有1个正整数 n(其中 $n \leq 26$),表示所给的虫蚀算式是 n 进制加法算式。其后3行中,每行有1个由 n 个大写英文字母组成的字符串,分别表示虫蚀算式中的2个加数及其和。相同的英文字母代表相同的数字。

★ **结果输出**

将计算出的虫蚀数字输出到文件 output.txt。在文件的第1行输出英文字母A,B,C,…,所表示的数字。

输入文件示例	输出文件示例
input.txt	output.txt
5	1 0 3 4 2
ABCED	
BDACE	
EBBAA	

分析与解答:

此题要找的是英文字母A,B,C,…与数字0,1,2,…的对应关系,其解空间显然是一棵排列树,可以套用搜索排列树的回溯法框架。

用类 Alpha 表示算法结构。

```
public class Alpha{
    static int n;
    static int [] x;
    static int [][] B;
    static node []cut;
}
```

其中,node是表示加法竖式的类。node中的3个数 a、b、c,表示加法竖式对应位的3个数。

```
public class node{
    int a,b,c;
    node next;
}
```

回溯法的主体是 backtrack。

```
static boolean backtrack(int i)
{
    if(i==n-1){
        if(oka()){
            for(int j=0;j<n;j++) System.out.print(x[j]+" ");
            System.out.println();return true;
        }
        else return false;
```

```
        }
        else
            for(int j=i;j<n;j++){
                MyMath.swap(x,i,j);
                if(constrain(i) && backtrack(i+1))return true;
                MyMath.swap(x,i,j);
            }
        return false;
}
```

oka 判断最终得到的算式是否成立。

```
static boolean oka()
{
    int carr=0;
    for(int i=n-1;i>=0;i--){
        int sum=x[B[i][0]]+x[B[i][1]]+carr;
        if(sum%n !=x[B[i][2]]) return false;
        carr=sum/n;
    }
    return true;
}
```

constrain 判断部分算式是否成立。

```
static boolean constrain(int i)
{
    for(int j=0;j<=i;j++) if(vio(cut[j]))return false;
    return true;
}
static boolean vio(node p)
{
    while(p!=null){
        if((x[p.a]+x[p.b])%n!=x[p.c]&&(x[p.a]+x[p.b]+1)%n!=x[p.c])return true;
        p=p.next;
    }
    return false;
}
```

compute 完成计算。

```
static void compute(int [][]Bb,int nn)
{
    n=nn;
    x=new int[n];
    cut=new node[n];
    B=Bb;
    construct();
```

```
    for(int i=0;i<n;i++) x[i]=i;
    backtrack(0);
}
```

construct 作结构初始化。

```
static void construct()
{
    for(int i=0;i<n;i++)cut[i]=null;
    for(int i=0;i<n;i++){
        int maxi=B[i][0];
        for(int j=1;j<3;j++) if(maxi<B[i][j])maxi=B[i][j];
        node p=new node();
        p.a=B[i][0];p.b=B[i][1];p.c=B[i][2];
        p.next=cut[maxi];
        cut[maxi]=p;
    }
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int [][]Bb=new int[n][3];
    for(int j=0;j<3;j++){
        String a=keyboard.readString();
        for(int i=0;i<n;i++) Bb[i][j]=a.charAt(i)-'A';
    }
    compute(Bb,n);
}
```

算法实现题 5-23　完备环序列问题

★ 问题描述

长度为 n 的环序列定义为含有 n 个互不相同的元素且首尾相接的环状序列。如果环序列中连续若干个数的和能形成一个连续的整数序列 $1,2,\cdots,m$，则称该环序列为一个完备的 (n,m) 序列。对于给定的 n，计算存在完备 (n,m) 序列的 m 的最大值。同时，计算有多少个不同的完备 (n,m) 序列。

★ 算法设计

对于给定的正整数 n，计算存在完备 (n,m) 序列的 m 的最大值；计算有多少个不同的完备 (n,m) 序列。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，$1\leqslant n\leqslant 10$。

★ 结果输出

将计算出的最大值 m 和不同的完备 (n,m) 序列的个数 k，以及所有不同的完备 (n,m) 序列输出到文件 output.txt。文件的第 1 行是 m 和 k；接下来的 k 行，每行是一个完备 (n,m)

序列。

输入文件示例	输出文件示例
input.txt	output.txt
2	3 1
	1 2

分析与解答：

长度为 n 的环序列最多可以产生 $r=n(n-1)+1$ 个不同的数，因此 m 的最大值不超过 r。用回溯法搜索所有可能的排列。

```
static void backtrack(int m)
{
    int y=r;
    for(int x=1;x<=m-1;x++) y-=a[x];
    for(int x=2;x<=y;x++)
        if(b[x]==0){
            a[m]=x;b[x]=1;
            if(m==n){if(oka()) ans();}
            else  backtrack(m+1);
            b[x]=0;
        }
}
```

oka 判断产生的整数是否为连续整数序列。maxi 记录产生的连续整数序列的长度。

```
static boolean oka()
{
    for(int i=1;i<=r;i++) t[i]=0;
    for(int i=1;i<=n;i++){
        int k=a[i];
        t[k]=1;
        for(int j=1;j<=n-1;j++){
            if(i+j<=n) k+=a[i+j];
            else k+=a[i+j-n];
            t[k]=1;
        }
    }
    for(int i=1;i<=r;i++)
        if(t[i]==0){if(maxi<i-1) maxi=i-1;return false;}
    maxi=r;
    return true;
}
```

ans 记录找到的解。

```
static void ans()
{
```

```
    for(int i=1;i<=n;i++) f[count][i]=a[i];
    count++;
}
```

init 作初始化计算。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    r=n*(n-1)+1;
    a=new int[n+1];
    b=new int[r+1];
    t=new int[r+1];
    f=new int[20][n+1];
    for(int i=0;i<=r;i++)b[i]=0;
    a[1]=1;b[1]=1;
}
```

out 输出所有解。

```
static void out()
{
    System.out.println(r+" "+count);
    for(int i=0;i<count;i++){
        for(int j=1;j<=n;j++) System.out.print(f[i][j]+" ");
        System.out.println();
    }
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    init();
    backtrack(2);
    if(maxi<r){r=maxi;backtrack(2);}
    out();
}
```

算法实现题 5-24　离散 01 串问题

★ 问题描述

(n,k)01 串定义为长度为 n 的 01 串，其中不含 k 个连续的相同子串。对于给定的正整数 n 和 k，计算 (n,k)01 串的个数。

★ 算法设计

对于给定的正整数 n 和 k，计算 (n,k)01 串的个数。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 k，$1\leqslant k,n\leqslant 40$。

★ 结果输出

将计算出的(n,k)01串的个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 3	4

分析与解答：

可看作子集选取问题，并套用子集树回溯搜索算法框架。由于对称性，只要考查首字符为0的情况，最后将找到的符合条件的01串个数加倍。

```
static void backtrack(int lev)
{
    if(lev>n) {tot+=2;return;}
    for(short i=0;i<2;i++){
        bstr[lev]=i;
        if(bstrok(lev))backtrack(lev+1);
    }
}
```

bstrok 判断当前子串是否满足要求。

```
static boolean bstrok(int lev)
{
    for(int i=0;i<k;i++) x[i]=lev-i;
    while (x[k-1]>0) {
        if(same()) return false;
        for(int i=0;i<k;i++) x[i]-=i+1;
    }
    return true;
}
```

same 判断当前子串是否重复。

```
static boolean same()
{
    int len=x[0]-x[1];
    for(int i=0;i<len;i++)
        for(int j=1;j<k;j++) if(bstr[x[j]+i]!=bstr[x[j-1]+i]) return false;
    return true;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
```

```
        k=keyboard.readInt();
        bstr=new short[n+1];
        x=new int[k];
        bstr[1]=0;
        if(k>1)backtrack(2);
        System.out.println(tot);
    }
```

算法实现题 5-25　喷漆机器人问题

★ 问题描述

F 大学开发出一种喷漆机器人 Rob，能用指定颜色给一块矩形材料喷漆。Rob 每次拿起一种颜色的喷枪，为指定颜色的小矩形区域喷漆。喷漆工艺要求，一个小矩形区域只能在所有紧靠它上方的矩形区域都喷过漆后，才能开始喷漆，且小矩形区域开始喷漆后必须一次性喷完，不能只喷一部分。为 Rob 编写一个自动喷漆程序，使 Rob 拿起喷枪的次数最少。

★ 算法设计

对于给定的矩形区域和指定的颜色，计算 Rob 拿起喷枪的最少次数。

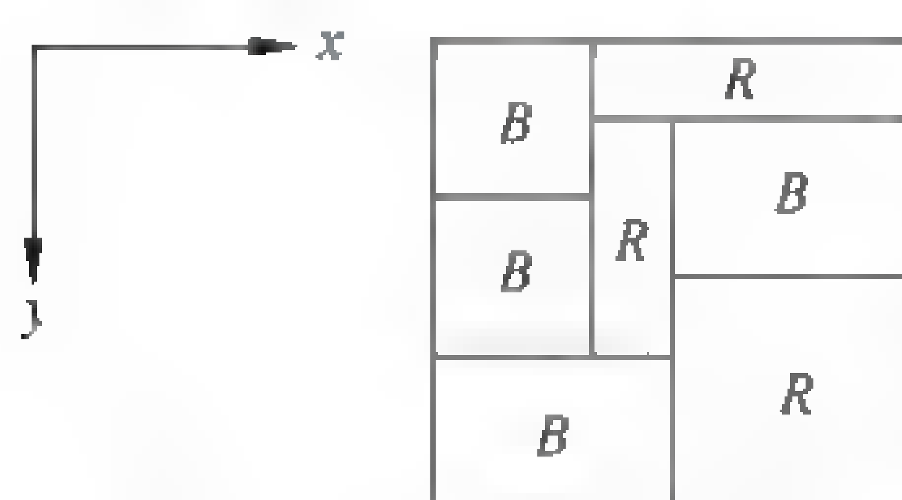


图 5-15　矩形喷漆材料

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，$1 \leqslant n \leqslant 16$，表示小矩形的个数。大矩形坐标系如图 5-15 所示，左上角点的坐标为(0,0)。颜色编号为正整数。接下来的 n 行，每行用 5 个整数 y_1, x_1, y_2, x_2, c 来表示一个矩形。(x_1, y_1)和(x_2, y_2)分别表示小矩形的左上角点坐标和右下角点坐标，c 表示小矩形的颜色。

★ 结果输出

将计算出的 Rob 拿起喷枪的最少次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7	3
0 0 2 2 1	
0 2 1 6 2	
2 0 4 2 1	
1 2 4 3 2	
1 3 3 6 1	
4 0 6 3 1	
3 3 6 6 2	

分析与解答：

用一个位向量表示每个小矩形的喷漆情况，对所有状态回溯搜索。

```
static void backtrack(int r,int p)
{
    if(m[r][p]>=0) return;
```

```
        for(int i=0;i<n; ++i)
            if(i!=r && ((p&po2[i])>0) && g[i][r]){
                m[r][p]=MAXx;
                return;
            }
        int np=p-po2[r];
        if(np==0)m[r][p]=1;
        else
            for(int i=0;i<n;++i)
                if((np&po2[i])>0){
                    backtrack(i,np);
                    int v=m[i][np]+(color[r]==color[i]? 0:1);
                    if(m[r][p]<0 || m[r][p]>v)m[r][p]=(short)v;
                }
    }
```

comp 执行回溯法，并输出最优值。

```
    static void comp()
    {
        for(int i=0;i<MAXn;i++)
            for(int j=0;j<MAX;j++)m[i][j]=-1;
        int opt=-1;
        for(int i=0;i<n;++i){
            backtrack(i,po2[n]-1);
            if(opt<0 || opt>m[i][po2[n]-1])
                opt=m[i][po2[n]-1];
        }
        System.out.println(opt);
    }
```

init 作初始化计算。

```
    static void init()
    {
        int [][]board=new int[MAXx][MAXy];
        for(int i=0;i<MAXx;i++)
            for(int j=0;j<MAXy;j++)board[i][j]=-1;
        ReadStream keyboard=new ReadStream();
        n=keyboard.readInt();
        for(int i=0;i<n; ++i){
            int x1=keyboard.readInt();
            int y1=keyboard.readInt();
            int x2=keyboard.readInt();
            int y2=keyboard.readInt();
            color[i]=keyboard.readInt();
            for(int j=x1;j<x2; ++j)
```

```
                for(int k=y1;k<y2;++k)board[j][k]=i;
        }
        for(int i=0;i<MAXn;i++)
            for(int j=0;j<MAXn;j++)g[i][j]=false;
        for(int i=0;i<MAXx;++i)
            for(int j=0;j<MAXy;++j)
                if(board[i][j]>=0 && board[i+1][j]>=0 && board[i][j]!=board[i+1][j])
                    g[board[i][j]][board[i+1][j]]=true;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    po2[0]=1;
    for(int i=1;i<MAXn;++i)po2[i]=po2[i-1]<<1;
    init();
    comp();
}
```

算法实现题 5-26　n^2-1 谜问题

★ 问题描述

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事"关羽义释曹操"而设计的智力玩具"华容道"，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1～8 按照任意次序排在 3×3 的方格阵列中，留下一个空格。与空格相邻的数字，允许从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1～8 按行排好序，如图 5-16 所示。在一般情况下，n^2-1 谜问题是将数字 $1\sim n^2-1$ 按照任意次序排在 $n\times n$ 的方格阵列中，留下一个空格。允许与空格相邻的数字从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将初始状态变换到目标状态。n^2-1 谜问题的目标状态是将数字 $1\sim n^2-1$ 按从小到大的次序排列，最后一个位置为空格。

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

图 5-16　重排九宫

★ 算法设计

对于给定的 $n\times n$ 方格阵列中数字 $1\sim n^2-1$ 初始排列，计算将初始排列通过合法移动变换为目标状态最少移动次数。

★ 数据输入

由文件 input.txt 给出输入数据。文件的第 1 行有 1 个正整数 n。接下来的 n 行是 $n\times n$ 方格阵列中的数字 $1\sim n^2-1$ 的初始排列，每行有 n 个数字表示该行方格中的数字，0 表示空格。

★ 结果输出

将计算出的最少移动次数和相应的移动序列输出到文件 output.txt。第 1 行是最少移动次数。从第 2 行开始，依次输出移动序列。

输入文件示例　　　　　　输出文件示例

input.txt	output.txt
3	2
1 2 3	5 8
4 0 6	
7 5 8	

分析与解答：

逐步深化的搜索算法描述如下：

```
static int rowsz,boardsz,moves,ppp;
static int []start;
static int []board;
static int []pos=new int[100];
static int row(int x) {return x/rowsz;}
static int col(int x) {return x%rowsz;}

static boolean solve(int l,int t,int p)
{
    int d,q,del;
    pos[l]=p;q=pos[l-1];
    if(t==0) {out(l,q);return true;}
    if(t>0)
        for(d=0;d<4;d++){
            del=trymove(p,q,d);p=ppp;
            if(del>0 && solve(l+1,t-del,p))return true;
            if(del>0)p=restore(p,d);
        }
    return false;
}

static void ids()
{
    int p=initstate();
    if(moves==0) {out(0,0);return;}
    while(!solve(1,moves,p))moves+=0x101;
}
```

上述算法描述中，trymove 按照可移动方向移动。

```
static int trymove(int p,int q,int d)
{
    int del=0;
    switch(d){
        case 0: // right
            if(col(p)<=rowsz-2&&q!=p+1){
                q=p+1;
                del=(col(board[q])<col(q)? 0x1:0x100);
```

```
                }
                break;
            case 1: // up
                if(row(p)>=1&&q!=p-rowsz){
                    q=p-rowsz;
                    del=(row(board[q])> row(q)? 0x1:0x100);
                }
                break;
            case 2: // left
                if(col(p)>=1&&q!=p-1){
                    q=p-1;
                    del=(col(board[q])> col(q)? 0x1:0x100);
                }
                break;
            case 3: // down
                if(row(p)<=rowsz-2&&q!=p+rowsz){
                    q=p+rowsz;
                    del=(row(board[q])<row(q)? 0x1:0x100);
                }
        }
        if(del>0){board[p]=board[q];p=q;}
        ppp=p;
        return del;
}
```

restore 恢复移动前的状态。

```
static int restore(int p,int d)
{
    int q=p-1;                  // right
    if(d==1) q=p+rowsz;         // up
    if(d==2) q=p+1;             // left
    if(d==3) q=p-rowsz;         // down
    board[p]=board[q];
    return q;
}
```

initstate 给出初始状态。

```
static int initstate()
{
    int j,del,p=0;
    for(j=moves=0;j<boardsz;j++)
        if(start[j]>=0){
            del=row(start[j])-row(j);
            moves+=(del<0? -del:del);
            del=col(start[j])-col(j);
```

```
            moves+=(del<0? -del:del);
        }
    pos[0]=boardsz;
    for(j=0;j<boardsz;j++){
        board[j]=start[j];
        if(board[j]<0) p=j;
    }
    return p;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    if(init())ids();
    else System.out.println("No solution!");
}
```

init 读入初始数据并判定可解性。

```
static boolean init()
{
    ReadStream keyboard=new ReadStream();
    rowsz=keyboard.readInt();
    boardsz=rowsz * rowsz;
    start=new int[boardsz];
    board=new int[boardsz];
    int del=0,t=0;
    for(int i=0;i<boardsz;i++){
        start[i]=keyboard.readInt();
        start[i]--;
        for(int s=0;s<i;s++)if(start[s]>start[i])del++;
        if(start[i]<0)t=i;
    }
    if(((row(t)+col(t)+del+rowsz)&0x1)==0) return false;
    return true;
}
```

out 输出最优解。

```
static void out(int l,int q)
{
    pos[l+1]=q;
    System.out.println((moves&0xff)+(moves>>8));
    if(moves>0){
        for(int j=0;j<boardsz;j++)board[j]=start[j];
        for(int k=1;k<l;k++){
            System.out.print((board[pos[k+1]]+1)+" ");
```

```
                board[pos[k]]=board[pos[k+1]];
            }
            System.out.println();
        }
}
```

上述算法可非递归化如下,效率明显提高。

```
static int rowsz,boardsz,moves;
static int []start;
static int []board;
static int []pos=new int[100];
static int []tim=new int[100];
static int []dir=new int[100];
static int row(int x){return x/rowsz;}
static int col(int x) {return x%rowsz;}

static void ids()
{
    int j,t=0,l=1,d=0,p=0,q=0,del=0,piece,moves,flag=0;
    moves=dist0();
    if(moves==0) {out(0,0,moves);return;}
    while(true){
        if(flag==0){
            t=moves;
            for(j=0;j<boardsz;j++){
                board[j]=start[j];
                if(board[j]<0)p=j;
            }
            pos[0]=boardsz;l=1;
            flag=1;
        }

    newlevel:
        if(flag==1){
            d=0;tim[l]=t;pos[l]=p;q=pos[l-1];flag=2;
        }

    trymove:
        if(flag==2){
            switch(d){
                case 0: // right
                    if(col(p)<=rowsz-2&&q!=p+1){
                        q=p+1;piece=board[q];
                        del=(col(piece)<col(q)? 0x1:0x100);
                        break;
                    }
                    d++;
```

```
            case 1: // up
                if(row(p)>=1&&q!=p  rowsz){
                    q=p-rowsz;piece=board[q];
                    del=(row(piece)> row(q)? 0x1:0x100);
                    break;
                }
                d++;
            case 2: // left
                if(col(p)>=1&&q!=p-1){
                    q=p-1;piece=board[q];
                    del=(col(piece)> col(q)? 0x1:0x100);
                    break;
                }
                d++;
            case 3: // down
                if(row(p)<=rowsz-2&&q!=p+rowsz){
                    q=p+rowsz;piece=board[q];
                    del=(row(piece)<row(q)? 0x1:0x100);
                    break;
                }
                d++;
            case 4: break; // goto backtrack;
        }
        if(d<4)flag=3;
        else flag=4;
    }
    if(flag==3){
        if(t<=del){
            if(t==del) {out(l,q,moves);return;}
            d++;flag=2; // goto trymove;
        }
        else{
            dir[l]=d;board[p]=board[q];t-=del;p=q;l++;
            flag=1; // goto newlevel;
        }
    }
backtrack:
    if(flag==4){
        if(l>1){
            l--;
            q=pos[l];board[p]=board[q];p=q;q=pos[l-1];t=tim[l];d=dir[l]+1;
            flag=2; // goto trymove;
        }
        else{moves+=0x101;flag=0;}
    }
```

```
        }
    }
```

其中,dist0 计算初始 Manhattan 距离。

```
static int dist0()
{
    int j,del,moves;
    for(j=moves=0;j<boardsz;j++)
        if(start[j]>=0){
            del=row(start[j])-row(j);
            moves+=(del<0? -del:del);
            del=col(start[j])-col(j);
            moves+=(del<0? -del:del);
        }
    return moves;
}
```

第 6 章 分支限界法

习题 6-1　0-1 背包问题的栈式分支限界法

栈式分支限界法将活结点表以后进先出(LIFO)的方式存储于一个栈中。试设计一个解 0-1 背包问题的栈式分支限界法，并说明栈式分支限界法与回溯法的区别。

分析与解答：

StackKnapsack 实施对子集树的栈式分支限界法。其中假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

算法中 enode 是当前扩展结点；cw 是该结点所相应的重量；cp 是相应的价值；up 是价值上界。算法的 while 循环不断扩展结点，首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点栈中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点栈。

具体算法描述如下：

```
private static double StackKnapsack()
{
  BBnode enode=null;
  int i=1;
  double bestp=0.0;
  double up=bound(1);
   while(true){
      // 检查当前扩展结点的左儿子结点
       double wt=cw+w[i];
       if(wt<=c){// 左儿子结点为可行结点
            if (cp+p[i] > bestp) bestp=cp+p[i];
            addLiveNode(up,cp+p[i],cw+w[i],i+1, enode, true);
       }
       up=bound(i+1);
       // 检查当前扩展结点的右儿子结点
       if (up >=bestp) // 右子树可能含最优解
            addLiveNode(up,cp,cw,i+1, enode, false);
       // 取下一扩展结点
       if(stk.empty()) return bestp;
```

```
            StkNode node=(StkNode) stk.pop();
            enode=node.liveNode;
            cw=node.weight;
            cp=node.profit;
            up=node.upperProfit;
            i=node.level;
        }
    }
```

其中,addLiveNode 将一个新的活结点插入到子集树和栈中。

```
private static void addLiveNode(double up, double pp, double ww, int lev, BBnode par, boolean ch)
{
  BBnode b=new BBnode(par, ch);
  StkNode node=new StkNode(b, up, pp, ww, lev);
  if(lev<=n)stk.push(node);
}
```

下面的算法 knapsack 完成对输入数据的预处理。主要任务是将各物品依其单位重量价值从大到小排好序。然后调用 StackKnapsack 完成对子集树的栈式分支限界搜索。

```
public static double knapsack(double [] pp, double [] ww, double cc)
{
  c=cc;
  n=pp.length-1;
  // 定义依单位重量价值排序的物品数组
  Element [] q=new Element [n];
  double ws=0.0;
  double ps=0.0;
  for (int i=1; i<=n; i++){
     q[i-1]=new Element(i, pp[i]/ww[i]);
     ps +=pp[i];
     ws +=ww[i];
  }
  if (ws<=c) return ps;
  // 依单位重量价值排序
  MergeSort.mergeSort(q);
  p=new double [n+1];
  w=new double [n+1];
  for (int i=1; i<=n; i++){
     p[i]=pp[q[n-i].id];
     w[i]=ww[q[n-i].id];
  }
  cw=0.0; cp=0.0;
  stk=new ArrayStack(1000);
  // 调用 StackKnapsack 求问题的最优解
  double maxp=StackKnapsack();
```

```
    return maxp;
}
```

栈式分支限界法与回溯法的主要不同在于对当前扩展结点所采用的扩展方式。在栈式分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。而回溯法中的结点有可能多次成为活结点。

习题 6-2　用最大堆存储活结点的优先队列式分支限界法

试修改解装载问题和解 0-1 背包问题的优先队列式分支限界法，使其仅使用一个最大堆来存储活结点，而不必存储所产生的解空间树。

分析与解答：

对于解 0-1 背包问题的优先队列式分支限界法，在堆结点中增加记录当前解的数组 x。

```
static class HeapNode implements Comparable
{
    double upperProfit;
    double profit;
    double weight;
    int level;
    int []x;

    HeapNode(double up, double pp, double ww, int lev, boolean ch, int []xx)
    {
        x=new int[n+1];
        for(int j=0;j<=n;j++)x[j]=xx[j];
        upperProfit=up;
        profit=pp;
        weight=ww;
        x[lev-1]=ch?1:0;
        level=lev;
    }

  HeapNode(){}

    public int compareTo(Object x)
    {
        double xup=((HeapNode) x).upperProfit;
        if (upperProfit<xup) return -1;
        if (upperProfit==xup) return 0;
        return 1;
    }
}
```

修改后的算法如下：

```
static double c;
static int n;
static double [] w;
static double [] p;
static double cw;
static double cp;
static int [] bestx;
static MaxHeap heap;
```

上界函数 bound 计算结点所相应价值的上界。

```
private static double bound(int i)
{// 计算结点所相应价值的上界
    double cleft=c-cw; // 剩余容量
    double b=cp;        // 价值上界
    // 以物品单位重量价值递减序装填剩余容量
    while (i<=n && w[i]<=cleft){
        cleft -= w[i];
        b += p[i];
        i++;
    }
    // 装填剩余容量装满背包
    if (i<=n) b += p[i]/w[i] * cleft;
    return b;
}
```

修改后的函数 addLiveNode 将一个新结点插入到优先队列中。

```
private static void addLiveNode(double up, double pp, double ww, int lev, boolean ch, int []x)
{
    HeapNode node=new HeapNode(up, pp, ww, lev, ch, x);
    heap.put(node);
}
```

maxKnapsack 实施对子集树的优先队列式分支限界搜索。其中假定各物品依其单位重量价值从大到小排好序。相应的排序过程可在算法的预处理部分完成。

```
private static double maxKnapsack()
{// 优先队列式分支限界法,返回最大价值,bestx 返回最优解
    int i=1;
    double bestp=0.0;
    double up=bound(1);
    HeapNode node=new HeapNode();
    while (i !=n+1){
        // 检查当前扩展结点的左儿子结点
        double wt=cw+w[i];
        if (wt<=c){// 左儿子结点为可行结点
            if (cp+p[i] > bestp) bestp=cp+p[i];
```

```
            addLiveNode(up,cp+p[i],cw+w[i],i+1, true,bestx);
        }
        up=bound(i+1);
        // 检查当前扩展结点的右儿子结点
        if (up >=bestp) addLiveNode(up,cp,cw,i+1, false,bestx); // 右子树可能含最优解
        // 取下一扩展结点
        node=(HeapNode) heap.removeMax();
        cw=node.weight;
        cp=node.profit;
        up=node.upperProfit;
        i=node.level;
        for(int j=0;j<=n;j++)bestx[j]=node.x[j];
    }
    // 构造当前最优解
    for(int j=0;j<=n;j++) bestx[j]=node.x[j];
    while(!heap.isEmpty()) heap.removeMax(); // 释放堆中所有结点
    return cp;
}
```

下面的算法 knapsack 完成对输入数据的预处理。主要任务是将各物品依其单位重量价值从大到小排好序。然后调用 maxKnapsack 完成对子集树的优先队列式分支限界搜索。

```
public static double knapsack(double [] pp, double [] ww, double cc, int [] xx)
{// 返回最大价值,bestx 返回最优解
    c=cc;
    n=pp.length-1;
    // 定义依单位重量价值排序的物品数组
    Element [] q=new Element [n];
    double ws=0.0;
    double ps=0.0;
    for (int i=1; i<=n; i++){
        q[i-1]=new Element(i, pp[i]/ww[i]);
        ps +=pp[i];
        ws +=ww[i];
    }
    if (ws<=c) {
        for (int i=1; i<=n; i++) xx[i]=1;
        return ps;
    }
    // 依单位重量价值排序
    MergeSort.mergeSort(q);
    p=new double [n+1];
    w=new double [n+1];
    for (int i=1; i<=n; i++){
        p[i]=pp[q[n-i].id];
        w[i]=ww[q[n-i].id];
```

```
    }
    cw=0.0;cp=0.0;
    bestx=new int [n+1];
    heap=new MaxHeap();
    // 调用 maxKnapsack 求问题的最优解
    double maxp=maxKnapsack();
    for (int i=1; i<=n; i++) xx[q[n-i].id]=bestx[i];
    return maxp;
}
```

习题 6-3　团顶点数的上界

解最大团问题的优先队列式分支限界法中，当前扩展结点满足 $cn+n-i \geq bestn$ 的右儿子结点被插入到优先队列中。如果将这个条件修改为满足 $cn+n-i>bestn$ 右儿子结点插入优先队列，仍能保证算法的正确性吗？为什么？

分析与解答：

如果将条件 $cn+n-i \geq bestn$ 修改为满足 $cn+n-i>bestn$ 右儿子结点插入优先队列，不能保证算法的正确性。因为在当前扩展结点处，团顶点数的上界为 $cn+n-i+1$。

习题 6-4　团顶点数改进的上界

考虑最大团问题的子集空间树中第 i 层的一个结点 x，设 MinDegree(x)是以结点 x 为根的子树中所有结点度数的最小值。

(1) 设 $x.u=\min\{x.cn+n-i+1, \text{MinDegree}(x)+1\}$，证明以结点 x 为根的子树中任一叶结点所相应的团的大小不超过 $x.u$。

(2) 依此 $x.u$ 的定义重写算法 bbMaxClique。

(3) 比较新旧算法所需的计算时间和产生的排列树结点数。

分析与解答：

(1) 在当前扩展结点 x 处，MinDegree(x)+1 显然是团顶点数的一个上界。主教材中在当前扩展结点 x 处团顶点数的上界为 $x.cn+n-i+1$。

由此可见，$\min\{x.cn+n-i+1, \text{MinDegree}(x)+1\}$ 是当前扩展结点 x 处团顶点数的上界。

(2) 在算法预处理时，先计算出每个顶点的 MinDegree(x)，然后在算法中修正团顶点数的上界。

(3) 新算法所产生的排列树结点数较少。

习题 6-5　修改解旅行售货员问题的分支限界法

试修改解旅行售货员问题的分支限界法，使得 $s=n-2$ 的结点不插入优先队列，而是将当前最优排列存储于 bestp 中。经这样修改后，算法在下一个扩展结点满足条件 $lcost \geq bestc$ 时结束。

分析与解答：

最小堆结点类型不变。

在算法中增加保存当前最优排列的数组 bestp。

修改后的算法描述如下：

```
public static float bbTSP(int v[])
{// 解旅行售货员问题的优先队列式分支限界法
    int n=v.length-1;
    MinHeap heap=new MinHeap();
    float [] minOut=new float [n+1];
    int [] bestp=new int [n+1];
    float minSum=0;
    // 计算 MinOut[i]=顶点 i 的最小出边费用
    for (int i=1; i<=n; i++){
        float min=Float.MAX_VALUE;
        for (int j=1; j<=n; j++)
            if (a[i][j]<Float.MAX_VALUE && a[i][j]<min) min=a[i][j];
        if (min==Float.MAX_VALUE) return Float.MAX_VALUE; // 无回路
        minOut[i]=min;
        minSum+=min;
    }
    int [] x=new int [n];
    for (int i=0; i<n; i++) x[i]=i+1;
    HeapNode enode=new HeapNode(0,0,minSum,0,x);
    float bestc=Float.MAX_VALUE;
    // 搜索排列空间树
    while (enode !=null && enode.s<n-1 && enode.lcost<bestc){
        x=enode.x;
        if (enode.s==n-2){// 当前扩展结点是叶结点的父结点
            // 再加 2 条边构成回路
            // 所构成回路是否优于当前最优解
            if (a[x[n-2]][x[n-1]]<Float.MAX_VALUE && a[x[n-1]][1]<
                Float.MAX_VALUE && enode.cc+a[x[n-2]][x[n-1]]+a[x[n-1]][1]<
                bestc){
                // 费用更小的回路
                bestc=enode.cc+a[x[n-2]][x[n-1]]+a[x[n-1]][1];
                for(int j=0;j<n;j++)bestp[j]=enode.x[j];
                enode.cc=bestc;
                enode.lcost=bestc;
                enode.s++;
                heap.put(enode);
            }
        }
        else{// 产生当前扩展结点的儿子结点
            for (int i=enode.s+1; i<n; i++)
                if (a[x[enode.s]][x[i]]<Float.MAX_VALUE){
                    float cc=enode.cc+a[x[enode.s]][x[i]];
                    float rcost=enode.rcost-minOut[x[enode.s]];
                    float b=cc+rcost;
                    if (b<bestc){// 子树可能含最优解
```

```
                        int [ ] xx=new int [n];
                        for (int j=0; j<n; j++) xx[j]=x[j];
                        xx[enode.s+1]=x[i];
                        xx[i]=x[enode.s+1];
                        HeapNode node=new HeapNode(b,cc,rcost,enode.s+1,xx);
                        heap.put(node);
                    }
                }
            }
        //取下一扩展结点
        enode=(HeapNode) heap.removeMin();
    }
    //将最优解复制到v[1:n]
    for(int i=0;i<n;i++) v[i+1]=bestp[i];
    return bestc;
}
```

习题 6-6　解旅行售货员问题的分支限界法中保存已产生的排列树

试修改解旅行售货员问题的分支限界法，使得算法保存已产生的排列树。

分析与解答：

排列树中结点类型定义如下：

```
static class BBnode
{
    BBnode parent;
    int s;
    int []x;
    BBnode(BBnode par, int ss,int []xx) {parent=par; s=ss; x=xx;}
}
```

最小堆结点类型定义如下：

```
private static class HeapNode implements Comparable
{
    float lcost, cc, rcost;
    BBnode ptr;
    HeapNode(BBnode node, float lc, float ccc, float rc)
    {ptr=node;lcost=lc;cc=ccc;}
    public int compareTo(Object x)
    {
        float xlc=((HeapNode) x).lcost;
        if (lcost<xlc) return -1;
        if (lcost==xlc) return 0;
        return 1;
    }
}
```

保存已产生的排列树的旅行售货员问题的分支限界法如下：

```
static float [][] a;
static MinHeap heap=new MinHeap();
public static float bbTSP(int v[])
{
    int n=v.length-1;
    float [] minOut=new float [n+1];
    float minSum=0;
    // 计算 minOut[i]=顶点 i 的最小出边费用
    for (int i=1; i<=n; i++){
        float min=Float.MAX_VALUE;
        for (int j=1; j<=n; j++)
            if (a[i][j]<Float.MAX_VALUE && a[i][j]<min) min=a[i][j];
        if (min==Float.MAX_VALUE) return Float.MAX_VALUE;
        minOut[i]=min;
        minSum+=min;
    }
    int [] x=new int [n];
    for (int i=0; i<n; i++) x[i]=i+1;
    addLiveNode(null,0,0,minSum,0,x);
    HeapNode enode=(HeapNode)heap.removeMin();
    float bestc=Float.MAX_VALUE;
    // 搜索排列空间树
    while (enode !=null && enode.ptr.s<n-1){
        x=enode.ptr.x;
        if (enode.ptr.s==n-2){
          if (a[x[n-2]][x[n-1]]<Float.MAX_VALUE && a[x[n-1]][1]<
             Float.MAX_VALUE && enode.cc+a[x[n-2]][x[n-1]]+
             a[x[n-1]][1]<bestc){
                bestc=enode.cc+a[x[n-2]][x[n-1]]+a[x[n-1]][1];
                addLiveNode(enode.ptr,bestc,bestc,enode.rcost,enode.ptr.s +1,x);
          }
        }
        else{// 产生当前扩展结点的儿子结点
            for (int i=enode.ptr.s+1; i<n; i++)
                if (a[x[enode.ptr.s]][x[i]]<Float.MAX_VALUE){
                    float cc=enode.cc+a[x[enode.ptr.s]][x[i]];
                    float rcost=enode.rcost-minOut[x[enode.ptr.s]];
                    float b=cc+rcost;
                    if (b<bestc){
                        // 子树可能含最优解
                        // 结点插入最小堆
                        int [] xx=new int [n];
                        for (int j=0; j<n; j++) xx[j]=x[j];
```

```
                        xx[enode.ptr.s+1]=x[i];
                        xx[i]=x[enode.ptr.s+1];
                        addLiveNode(enode.ptr,b,cc,rcost,enode.ptr.s+1,xx);
                    }
                }
            }
            enode=(HeapNode) heap.removeMin(); // 取下一扩展结点
        }
        for (int i=0; i<n; i++) v[i+1]=x[i];
        return bestc;
    }
```

其中,addLiveNode将新产生的活结点加入到排列树中,并将这个新结点插入到表示活结点优先队列的最小堆中。

```
    private static void addLiveNode(BBnode par, float lc, float ccc, float rc, int ss, int [] xx)
    {
        BBnode b=new BBnode(par,ss,xx);
        HeapNode node=new HeapNode(b,lc,ccc,rc);
        heap.put(node);
    }
```

习题6-7　电路板排列问题的队列式分支限界法

试设计解电路板排列问题的队列式分支限界法,并使算法在运行结束时输出最优解和最优值。

分析与解答:

```
    static class BoardNode{
        int s,cd;
        int []x;
        int []now;

        BoardNode(int cdd, int [] noww, int ss, int [] xx) {
            cd=cdd;
            now=noww;
            s=ss;
            x=xx;
        }
    }
```

解电路板排列问题的队列式分支限界法实现如下:

```
    public static int fifoBoards(int [][] board, int m, int [] bestx)
    {// 解电路板排列问题的队列式分支限界法
        int n=board.length-1;
        ArrayQueue queue=new ArrayQueue(); // 活结点队列
        BoardNode enode=new BoardNode(0, new int [m+1], 0, new int [n+1]);
        // now[i]=x[1:s]所含连接块i中电路板数
```

```
    // total[i]=连接块 i 中电路板数
    int [] total=new int [m+1];
    for (int i=1; i<=n; i++){
        enode.x[i]=i; // 初始排列为 123…n
        for (int j=1; j<=m; j++) total[j] +=board[i][j]; // 连接块 j 中电路板数
    }
    int bestd=m+1; // 当前最小密度
    int [] x=null;
    while(true){
        if (enode.s==n-1){// 仅一个儿子结点
            int ld=0; // 最后一块电路板的密度
            for (int j=1; j<=m; j++) ld +=board[enode.x[n]][j];
            if (ld<bestd && enode.cd<bestd){// 密度更小的电路板排列
                x=enode.x;
                bestd=Math.max(ld, enode.cd);
            }
        }
        else{// 产生当前扩展结点的所有儿子结点
            for (int i=enode.s+1; i<=n; i++){
                BoardNode node=new BoardNode(0, new int [m+1], 0, new int [n+1]);
                for (int j=1; j<=m; j++)
                    node.now[j]=enode.now[j]+board[enode.x[i]][j];// 新插入的电路板
                int ld=0; // 新插入电路板的密度
                for (int j=1; j<=m; j++)
                    if (node.now[j] > 0 && total[j] !=node.now[j]) ld++;
                node.cd=Math.max(ld, enode.cd);
                if (node.cd<bestd){// 可能产生更好的叶结点
                    node.s=enode.s+1;
                    for (int j=1; j<=n; j++) node.x[j]=enode.x[j];
                    node.x[node.s]=enode.x[i];
                    node.x[i]=enode.x[node.s];
                    queue.put(node);
                }
            }
        }
        if(queue.isEmpty()) break;
        else enode=(BoardNode) queue.remove(); // 取下一扩展结点
    }
    for (int i=1; i<=n; i++) bestx[i]=x[i];
    return bestd;
}
```

算法实现题 6-1　最小长度电路板排列问题(一)

★ 问题描述

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法

是,将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1,2,\cdots,n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2,\cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中,每个连接块 N_i 是 B 的一个子集,且 N_i 中的电路板用同一根导线连接在一起。

例如,设 $n=8,m=5$。给定 n 块电路板及其 m 个连接块如下:

$B=\{1,2,3,4,5,6,7,8\}$; $L=\{N_1, N_2, N_3, N_4, N_5\}$;

$N_1=\{4,5,6\}$; $N_2=\{2,3\}$; $N_3=\{1,3\}$; $N_4=\{3,6\}$; $N_5=\{7,8\}$。

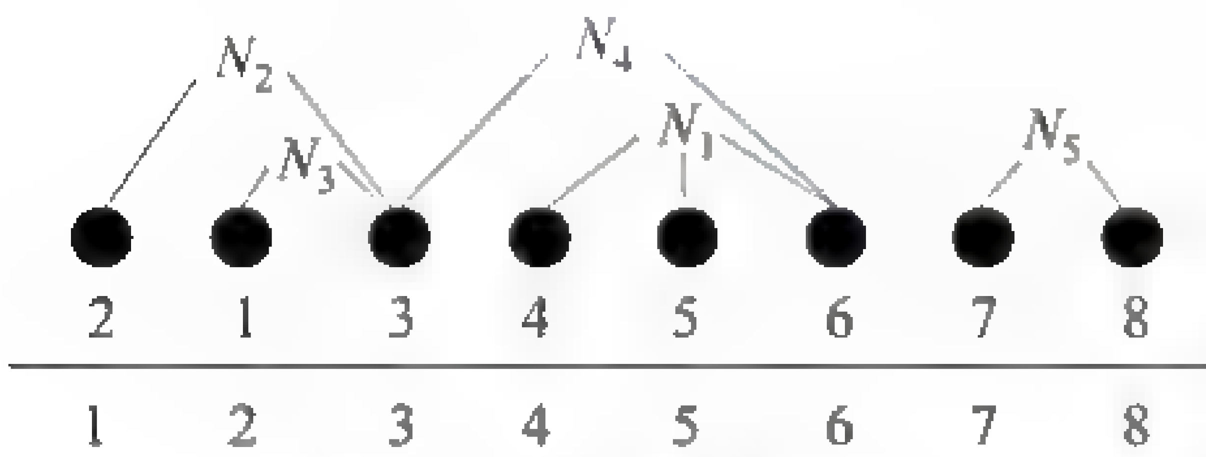


图 6-1 电路板排列

这 8 块电路板的一个可能的排列如图 6-1 所示。

在最小长度电路板排列问题中,连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如在图 6-1 所示的电路板排列中,连接块 N_4 的第 1 块电路板在插槽 3 中,它的最后 1 块电路板在插槽 6 中,因此 N_4 的长度为 3。同理,N_2 的长度为 2。图 6-1 中连接块最大长度为 3。试设计一个队列式分支限界法找出所给 n 个电路板的最佳排列,使得 m 个连接块中最大长度达到最小。

★ 算法设计

对于给定的电路板连接块,设计一个队列式分支限界法,找出所给 n 个电路板的最佳排列,使得 m 个连接块中最大长度达到最小。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m (其中 $1\leqslant m,n\leqslant 20$)。接下来的 n 行中,每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中,1 表示电路板 k 在连接块 j 中。

★ 结果输出

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第 1 行是最小长度;第 2 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

★ **评分**

未按照题目要求用队列式分支限界法解题，则所得分数减半。

分析与解答：

与最小密度电路板排列问题类似，结点元素类型是 BoardNode。

```
static class BoardNode{
    int s,cd;
    int []x;
    int []low;
    int []high;
    BoardNode(int cdd, int ss, int [] loww, int [] highh, int [] xx) {
        cd=cdd; s=ss;
        low=loww;high=highh; x=xx;
    }
    int len(){
        int tmp=0;
        for(int k=1;k<=m;k++)
            if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k])tmp=high[k]-low[k];
        return tmp;
    }
}
```

其中，len 计算当前排列的最小长度。

解最小长度电路板排列问题的队列式分支限界法如下：

```
public static int fifoBoards(int [][] board, int m, int [] bestx)
{
    int n=board.length-1;
    ArrayQueue queue=new ArrayQueue();
    BoardNode enode=new BoardNode(0,0,new int [m+1],new int [m+1], new int [n+1]);
    for(int i=1;i<=m;i++){enode.high[i]=0;enode.low[i]=n+1;}
    for (int i=1; i<=n; i++)enode.x[i]=i;
    int bestd=n+1;
    while(true){
        if (enode.s==n-1){
            for(int j=1;j<=m;j++)
                if(board[enode.x[n]][j]>0 && n>enode.high[j])enode.high[j]=n;
            int ld=enode.len();
            if(ld<bestd){
                bestd=ld;
                for (int i=1; i<=n; i++) bestx[i]=enode.x[i];
            }
        }
        else{
            int curr=enode.s+1;
            for(int i=enode.s+1;i<=n;i++){
```

```
                BoardNode node=new BoardNode(0,0,
                    new int [m+1],new int [m+1], new int [n+1]);
                for(int j=1;j<=m;j++){
                    node.low[j]=enode.low[j];node.high[j]=enode.high[j];
                    if(board[enode.x[i]][j]>0){
                        if(curr<node.low[j])node.low[j]=curr;
                        if(curr>node.high[j])node.high[j]=curr;
                    }
                }
                node.cd=node.len();
                if(node.cd<bestd){
                    node.s=enode.s+1;
                    for(int j=1;j<=n;j++) node.x[j]=enode.x[j];
                    node.x[node.s]=enode.x[i];
                    node.x[i]=enode.x[node.s];
                    queue.put(node);
                }
            }
        }
        if(queue.isEmpty()) break;
        else enode=(BoardNode) queue.remove();
    }
    return bestd;
}
```

算法实现题 6-2 最小长度电路板排列问题(二)

★ 问题描述

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1,2,\cdots,n\}$ 是 n 块电路板的集合。集合 $L=\{N_1,N_2,\cdots,N_m\}$ 是 n 块电路板的 m 个连接块。其中，每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。

例如，设 $n=8,m=5$。给定 n 块电路板及其 m 个连接块如下：

$B=\{1,2,3,4,5,6,7,8\}$；$L=\{N_1, N_2, N_3, N_4, N_5\}$；

$N_1=\{4,5,6\}$；$N_2=\{2,3\}$；$N_3=\{1,3\}$；$N_4=\{3,6\}$；$N_5=\{7,8\}$。

这 8 块电路板的一个可能的排列如图 6-2 所示。

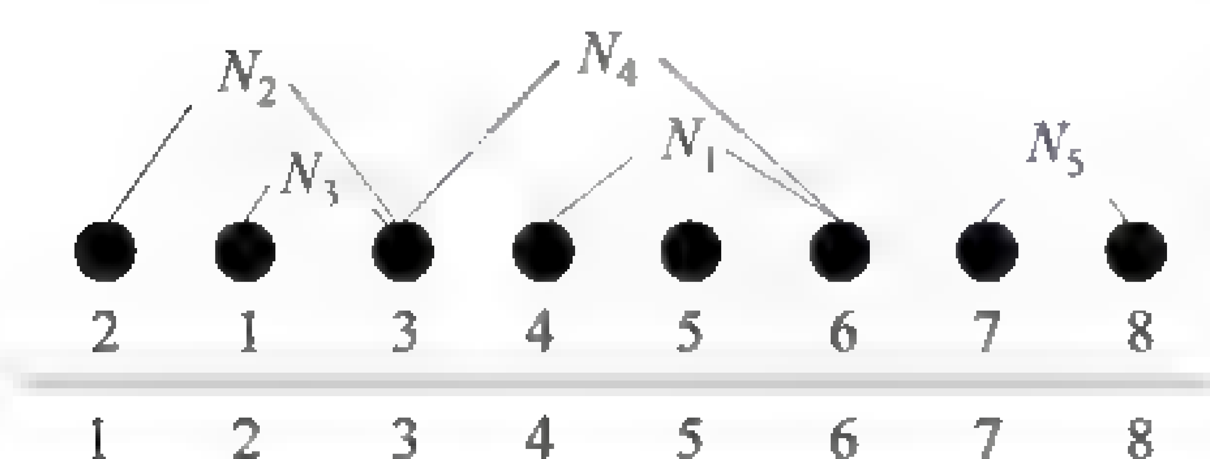


图 6-2 最小长度电路板排列

在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，在图 6-2 所示的电路板排列中，连接块 N_4 的第 1 块电路板在插槽 3 中，它的最后 1 块电路板在插槽 6 中，因此，N_4 的长度为 3。同理，N_2 的长度为 2。图 6-2 中连接块最大长

度为3。试设计一个优先队列式分支限界法找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

★ **算法设计**

对于给定的电路板连接块，设计一个优先队列式分支限界法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 m（其中 $1\leqslant m,n\leqslant 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为0表示电路板 k 不在连接块 j 中，1表示电路板 k 在连接块 j 中。

★ **结果输出**

将计算出的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第1行是最小长度；接下来的1行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

★ **评分**

未按照题目要求用优先队列式分支限界法解题，则所得分数减半。

分析与解答：

与最小密度电路板排列问题类似，堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int s,cd;
    int []x;
    int []low;
    int []high;
    HeapNode(int cdd, int ss, int [] loww, int [] highh, int [] xx) {
        cd=cdd;
        s=ss;
        low=loww;
        high=highh;
        x=xx;
    }
    int len(){
```

```
            int tmp=0;
            for(int k=1;k<=m;k++)
                if(low[k]<=n && high[k]>0 && tmp<high[k]-low[k])tmp=high[k]-low[k];
            return tmp;
        }
        public int compareTo(Object x) {
            int xcd=((HeapNode) x).cd;
            if (cd<xcd) return -1;
            if (cd==xcd) return 0;
            return 1;
        }
    }
```

其中,len计算当前排列的最小长度。

解最小长度电路板排列问题的优先队列式分支限界法如下:

```
public static int pfBoards(int [][] board, int m, int [] bestx)
{
    int n=board.length-1;
    MinHeap heap=new MinHeap();
    HeapNode enode=new HeapNode(0,0,new int [m+1],new int [m+1], new int [n+1]);
    for(int i=1;i<=m;i++){enode.high[i]=0;enode.low[i]=n+1;}
    for (int i=1; i<=n; i++)enode.x[i]=i;
    int bestd=n+1;
    do{
        if (enode.s==n-1){
            for(int j=1;j<=m;j++)
                if(board[enode.x[n]][j]>0 && n>enode.high[j])enode.high[j]=n;
            int ld=enode.len();
            if(ld<bestd){
                bestd=ld;
                for (int i=1; i<=n; i++) bestx[i]=enode.x[i];
            }
        }
        else{
            int curr=enode.s+1;
            for(int i=enode.s+1;i<=n;i++){
                HeapNode node=new HeapNode(0,0,
                                new int [m+1],new int [m+1], new int [n+1]);
                for(int j=1;j<=m;j++){
                    node.low[j]=enode.low[j];node.high[j]=enode.high[j];
                    if(board[enode.x[i]][j]>0){
                        if(curr<node.low[j])node.low[j]=curr;
                        if(curr>node.high[j])node.high[j]=curr;
                    }
                }
                node.cd=node.len();
```

```
                if(node.cd<bestd){
                    node.s=enode.s+1;
                    for(int j=1;j<=n;j++) node.x[j]=enode.x[j];
                    node.x[node.s]=enode.x[i];
                    node.x[i]=enode.x[node.s];
                    heap.put(node);
                }
            }
        }
        enode=(HeapNode) heap.removeMin();
    } while (enode !=null && enode.cd<bestd);
    return bestd;
}
```

算法实现题 6-3　最小权顶点覆盖问题

★ 问题描述

给定一个赋权无向图 $G=(V,E)$，每个顶点 $v\in V$ 都有一个权值 $w(v)$。如果 $U\subseteq V$，且对任意 $(u,v)\in E$ 有 $u\in U$ 或 $v\in U$，就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

★ 算法设计

对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 $1,2,\cdots,n$。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中，每行有 2 个正整数 u,v，表示图 G 的一条边 (u,v)。

★ 结果输出

将计算出的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt 中。文件的第 1 行是最小权顶点覆盖顶点权之和；文件第 2 行是最优解 x_i，$1\leqslant i\leqslant n$，$x_i=0$ 表示顶点 i 不在最小权顶点覆盖中，$x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

输入文件示例	输出文件示例
input.txt	output.txt
7 7	13
1 100 1 1 1 100 10	1 0 1 1 0 0 1
1 6	
2 4	
2 5	
3 6	
4 5	
4 6	
6 7	

★ 评分

未按照题目要求用优先队列式分支限界法解题，则所得分数减半。

分析与解答:

最小堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int i,cn;
    int []x;
    int []c;
    HeapNode(int ii, int cnn, int [] cc, int [] xx)
    {i=ii;cn=cnn;c=cc;x=xx;}
    public int compareTo(Object x) {
        int xcn=((HeapNode) x).cn;
        if (cn<xcn) return -1;
        if (cn==xcn) return 0;
        return 1;
    }
}
```

解最小权顶点覆盖问题的优先队列式分支限界法如下:

```
static int n,bestn;
static int []w;
static int []bestx;
static int [][]a;
static MinHeap heap=new MinHeap();
static void bbVC()
{
    HeapNode E=new HeapNode(1,0,new int[n+1], new int[n+1]);
    for(int j=1;j<=n;j++){E.x[j]=E.c[j]=0;}
    int i=1,cn=0;
    while(true){
        if(i>n){
            if(cover(E)){
                for(int j=1;j<=n;j++) bestx[j]=E.x[j];
                bestn=cn;
                break;
            }
        }
        else{
            if(!cover(E)) addLiveNode(E,cn,i,true);
            addLiveNode(E,cn,i,false);}
        if(heap.isEmpty()) break;
        E=(HeapNode)heap.removeMin();
        cn=E.cn;
        i=E.i+1;
    }
}
```

cover 判定图是否已完全覆盖。

```
static boolean cover(HeapNode E)
{
    for(int j=1;j<=n;j++)if(E.x[j]==0 && E.c[j]==0)return false;
    return true;
}
```

addLiveNode 将活结点加入堆中。

```
private static void addLiveNode(HeapNode E,int cn,int i,boolean ch)
{
    HeapNode N=new HeapNode(i,cn,new int[n+1], new int[n+1]);
    for(int j=1;j<=n;j++){N.x[j]=E.x[j];N.c[j]=E.c[j];}
    N.x[i]=ch?1:0;
    if(ch){
        N.cn=cn+w[i];
        for(int j=1;j<=n;j++)if(a[i][j]>0)N.c[j]++;
    }
    else N.cn=cn;
    N.i=i;
    heap.put(N);
}
```

minCover 完成最小覆盖计算。

```
public static int minCover(int [][]aa,int []v,int nn)
{
    w=new int [n+1];
    for(int j=1;j<=n;j++){w[j]=v[j];}
    a=aa;n=nn;bestx=v;
    bbVC();
    return bestn;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
   ReadStream keyboard=new ReadStream();
   n=keyboard.readInt();
   int e=keyboard.readInt();
   int [][] a=new int [n+1][n+1];
   for(int i=0;i<=n;i++)
       for(int j=0;j<=n;j++)a[i][j]=0;
   int [] p=new int[n+1];
   for(int i=1;i<=n;i++)p[i]=keyboard.readInt();
   for(int i=1;i<=e;i++){
       int u=keyboard.readInt();
       int v=keyboard.readInt();
```

```
            a[u][v]=1;a[v][u]=1;
        }
        System.out.println(minCover(a,p,n));
        for (int i=1; i<=n; i++) System.out.print(p[i]+" ");
        System.out.println();
    }
```

算法实现题6-4 无向图的最大割问题

★ 问题描述

给定一个无向图 $G=(V,E)$，设 $U \subseteq V$ 是 G 的顶点集。对任意 $(u,v)\in E$，若有 $u\in U$ 且 $v\in V-U$，就称 (u,v) 为关于顶点集 U 的一条割边。顶点集 U 的所有割边构成图 G 的一个割。G 的最大割是指 G 中所含边数最多的割。

★ 算法设计

对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最大割。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 $1,2,\cdots,n$。接下来的 m 行中，每行有2个正整数 u,v，表示图 G 的一条边 (u,v)。

★ 结果输出

将计算出的最大割的边数和顶点集 U 输出到文件 output.txt 中。文件的第1行是最大割的边数；文件的第2行是表示顶点集 U 的向量，x_i，$1\leqslant i\leqslant n$，$x_i=0$ 表示顶点 i 不在顶点集 U 中，$x_i=1$ 表示顶点 i 在顶点集 U 中。

输入文件示例	输出文件示例
input.txt	output.txt
7 18	12
1 4	1 1 1 0 1 0 0
1 5	
1 6	
1 7	
2 3	
2 4	
2 5	
2 6	
2 7	
3 4	
3 5	
3 6	
3 7	
4 5	
4 6	
5 6	
5 7	
6 7	

★ **评分**

未按照题目要求用优先队列式分支限界法解题，则所得分数减半。

分析与解答：

最大堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int i,cut,e;
    int []x;
    HeapNode(int ii, int cutt, int ee, int [] xx)
    {i=ii;cut=cutt;e=ee;x=xx;}
    public int compareTo(Object x) {
        int xcn=((HeapNode) x).cut+((HeapNode) x).e;
        if (cut+e<xcn) return -1;
        if (cut+e==xcn) return 0;
        return 1;
    }
}
```

解无向图最大割问题的优先队列式分支限界法如下：

```
static int n,e,bestn;
static int []bestx;
static int [][]a;
static MaxHeap heap=new MaxHeap();

static void bbCut()
{
    HeapNode E=new HeapNode(1,0,e, new int[n+1]);
    for(int j=1;j<=n;j++)E.x[j]=0;
    while(true){
        if(E.i>n){
            if(E.cut>bestn){
                for(int j=1;j<=n;j++) bestx[j]=E.x[j];
                bestn=E.cut;
            }
        }
        else{
            addLiveNode(E,true);
            if(E.cut+E.e>bestn) addLiveNode(E,false);
        }
        if(heap.isEmpty()) break;
        E=(HeapNode)heap.removeMax();
    }
}
```

addLiveNode 将活结点加入堆中。

```
private static void addLiveNode(HeapNode E,boolean ch)
```

```
{
    int i=E.i;
    HeapNode N=new HeapNode(i,E.cut,E.e, new int[n+1]);
    for(int j=1;j<=n;j++)N.x[j]=E.x[j];
    N.x[i]=ch?1:0;
    if(ch){
        for(int j=1;j<=n;j++)
            if(a[i][j]>0){
                if(N.x[j]==0){N.cut++;N.e--;}
                else N.cut--;
            }
    }
    N.i=i+1;
    heap.put(N);
}
```

maxCut 完成最大割计算。

```
public static int maxCut(int [][]aa,int []v,int nn,int ee)
{
    a=aa;n=nn;e=ee;bestn=0;bestx=v;
    bbCut();
    return bestn;
}
```

算法的主函数如下:

```
public static void main(String [] args)
{
   ReadStream keyboard=new ReadStream();
   n=keyboard.readInt();
   e=keyboard.readInt();
   int [][] a=new int [n+1][n+1];
   for(int i=0;i<=n;i++)
       for(int j=0;j<=n;j++)a[i][j]=0;
   int []p=new int[n+1];
   for(int i=1;i<=e;i++){
       int u=keyboard.readInt();
       int v=keyboard.readInt();
       a[u][v]=1;a[v][u]=1;
   }
   System.out.println(maxCut(a,p,n,e));
   for (int i=1; i<=n; i++) System.out.print(p[i]+" ");
   System.out.println();
}
```

算法实现题 6-5　最小重量机器设计问题

★ 问题描述

设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。

设计一个优先队列式分支限界法，给出总价格不超过 d 的最小重量机器设计。

★ 算法设计

对于给定的机器部件重量和机器部件价格，设计一个优先队列式分支限界法，计算总价格不超过 d 的最小重量机器设计。

★ 数据输入

由文件 input. txt 给出输入数据。第 1 行有 3 个正整数 n，m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

★ 结果输出

将计算出的最小重量，以及每个部件的供应商输出到文件 output. txt。

输入文件示例	输出文件示例
input. txt	output. txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

★ 评分

如果没有按照题目要求用分支限界法解题，则所得分数减半。

分析与解答：

状态空间树中结点类型为 bbnode。

```
static class bbnode {
    bbnode parent;
    int mj;
    bbnode(bbnode par, int j) {parent=par;mj=j;}
}
```

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int profit;
    int weight;
    int level;
    bbnode ptr;
    HeapNode(bbnode pt,int p, int w, int l)
    {ptr=pt;profit=p;weight=w;level=l;}
```

```
    public int compareTo(Object x) {
        int xw=((HeapNode) x).weight;
        if (weight<xw) return -1;
        if (weight==xw) return 0;
        return 1;
    }
}
```

解最小重量机器设计问题的优先队列式分支限界法如下：

```
static bbnode E=new bbnode (null,0);
static int n,m,cc,cp,cw;
static int []bestx;
static int [][]w;
static int [][]c;
static MinHeap heap=new MinHeap();

static int minWeightMachine()
{
    bestx=new int[n+1];
    int i=1;
    cw=cp=0;
    int besint=0;
    while (i!=n+1){
        for(int j=1;j<=m;j++){
            int wt=cw+w[i][j];
            int ct=cp+c[i][j];
            if (ct<=cc) addLiveNode(ct,wt,i+1,j);
        }
        if(heap.isEmpty()) break;
        HeapNode N=(HeapNode)heap.removeMin();
        E=N.ptr;cw=N.weight;
        cp=N.profit;i=N.level;
    }
    if(i<=n) return 0;
    for (int j=n;j>0;j--){
        bestx[j]=E.mj;
        E=E.parent;
    }
    return cw;
}
```

addLiveNode 将活结点加入堆中。

```
private static void addLiveNode(int cp,int cw,int i,int j)
{
    bbnode b=new bbnode (E,j);
    HeapNode N=new HeapNode (b,cp,cw,i);
```

```
    heap.put(N);
}
```

machine 完成最小重量机器设计。

```
public static int machine(int [][]cx,int [][]wx,int ccx,int nx,int mx,int []best)
{
    c=cx;w=wx;cp=0;cw=0;cc=ccx;
    n=nx;m=mx;bestx=best;
    int besp=minWeightMachine();
    for (int j=1;j<=n;j++) best[j]=bestx[j];
    return besp;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
   ReadStream keyboard=new ReadStream();
   int n=keyboard.readInt();
   int m=keyboard.readInt();
   int cc=keyboard.readInt();
   int [][]c=new int[n+1][m+1];
   int [][]w=new int[n+1][m+1];
   int []bestx=new int[n+1];
   for(int i=1;i<=n;i++)
      for(int j=1;j<=m;j++)c[i][j]=keyboard.readInt();
   for(int i=1;i<=n;i++)
      for(int j=1;j<=m;j++)w[i][j]=keyboard.readInt();
   int answer=machine(c,w,cc,n,m,bestx);
   if(answer>0){
      System.out.println(answer);
      for(int i=1;i<=n;i++)System.out.print(bestx[i]+" ");
      System.out.println();
   }
   else System.out.println("No Solution!");
}
```

算法实现题 6-6　运动员最佳匹配问题

★ 问题描述

羽毛球队有男女运动员各 n 人。给定两个 $n\times n$ 矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$。$P[i][j]$是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势；$Q[i][j]$是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响，$P[i][j]$不一定等于 $Q[j][i]$。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j]*Q[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

★ **算法设计**

设计一个优先队列式分支限界法,对于给定的男女运动员竞赛优势,计算男女运动员最佳配对法,使各组男女双方竞赛优势的总和达到最大。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有1个正整数 n(其中 $1\leqslant n\leqslant 20$)。接下来的 $2n$ 行,每行 n 个数。前 n 行是 p,后 n 行是 q。

★ **结果输出**

将计算出的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	
3 4 5	
2 2 2	
3 5 3	
4 5 1	

★ **评分**

如果没有按照题目要求用分支限界法解题,则所得分数减半。

分析与解答:

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int s,val;
    int []r;
    HeapNode(int ss, int v, int []rr)
    {
        s=ss;val=v;r=rr;
        for(int i=1;i<=n;i++) r[i]=i;
    }
    void compute(int ii)
    {
        int temp=0;
        for(int i=1;i<=ii;i++) temp+=p[i][r[i]] * q[r[i]][i];
        val=temp;
    }
    public int compareTo(Object x) {
        int xv=((HeapNode) x).val;
        if (val<xv) return -1;
        if (val==xv) return 0;
        return 1;
    }
```

```
}
```

解运动员最佳匹配问题的优先队列式分支限界法如下：

```
static int n,best=0;
static int []bestr;
static int [][]p;
static int [][]q;
static MaxHeap heap=new MaxHeap();

static int getbest()
{
    HeapNode E=new HeapNode (0,0,new int[n+1]);
    while(true){
        if(E.s==n-1){
            E.compute(n);
            if(E.val>best){bestr=E.r;best=E.val;}
        }
        else{
            for(int i=E.s+1;i<=n;i++){
                HeapNode N=new HeapNode (E.s+1,E.val,new int[n+1]);
                for(int j=1;j<=n;j++)N.r[j]=E.r[j];
                N.r[N.s]=E.r[i];
                N.r[i]=E.r[N.s];
                N.compute(N.s);
                heap.put(N);
            }
        }
        if(heap.isEmpty()) return best;
        E=(HeapNode)heap.removeMax();
    }
}
```

算法实现题 6-7　n 后问题

★ 问题描述

在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n 后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后，任何 2 个皇后不放在同一行或同一列或同一斜线上。

★ 算法设计

设计一个解 n 后问题的队列式分支限界法，计算在 $n \times n$ 个方格上放置彼此不受攻击的 n 个皇后的一个放置方案。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

★ 结果输出

将计算出的彼此不受攻击的 n 个皇后的一个放置方案输出到文件 output.txt。文件的

第1行是 n 个皇后的放置方案。

输入文件示例	输出文件示例
input.txt	output.txt
5	1 3 5 2 4

★ 评分

如果没有按照题目要求用分支限界法解题,则所得分数减半。

分析与解答:

排列空间树中结点类型为 qNode。

```
static class qNode{
    int i;
    int []x;
    qNode(int ii, int [] xx) {
        i=ii;x=xx;
        for(int j=1;j<=n;j++)x[j]=j;
    }
    boolean constrain()
    {
        for(int j=1;j<i;j++)
            if((Math.abs(i-j)==Math.abs(x[j]-x[i])) || (x[j]==x[i]))return false;
        return true;
    }
}
```

用队列式分支限界法搜索 n 皇后问题排列树的算法 fifoQueen 如下:

```
static void fifoQueen()
{
    ArrayQueue queue=new ArrayQueue();
    qNode E=new qNode (0, new int[n+1]);
    bestx=new int[n+1];
    found=false;
    while(true){
        if(E.i==n){
            for(int k=1;k<=n;k++)bestx[k]=E.x[k];
            found=true;
        }
        else
            for(int i=E.i+1;i<=n;i++){
                qNode N=new qNode (E.i+1, new int[n+1]);
                for(int j=1;j<=n;j++) N.x[j]=E.x[j];
                N.x[N.i]=E.x[i];
                N.x[i]=E.x[N.i];
                if(N.constrain())queue.put(N);
            }
```

```
        if(queue.isEmpty()) break;
        else E=(qNode) queue.remove();
    }
    for (int i=1; i<=n; i++) System.out.print(bestx[i]+" ");
    System.out.println();
}
```

算法实现题 6-8　圆排列问题

★ 问题描述

给定 n 个大小不等的圆 $c_1, c_2, \cdots, c_n$，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 $n=3$，且所给的 3 个圆的半径分别为 1,1,2 时，这 3 个圆的最小长度的圆排列如图 6-3 所示。其最小长度为 $2+4\sqrt{2}$。

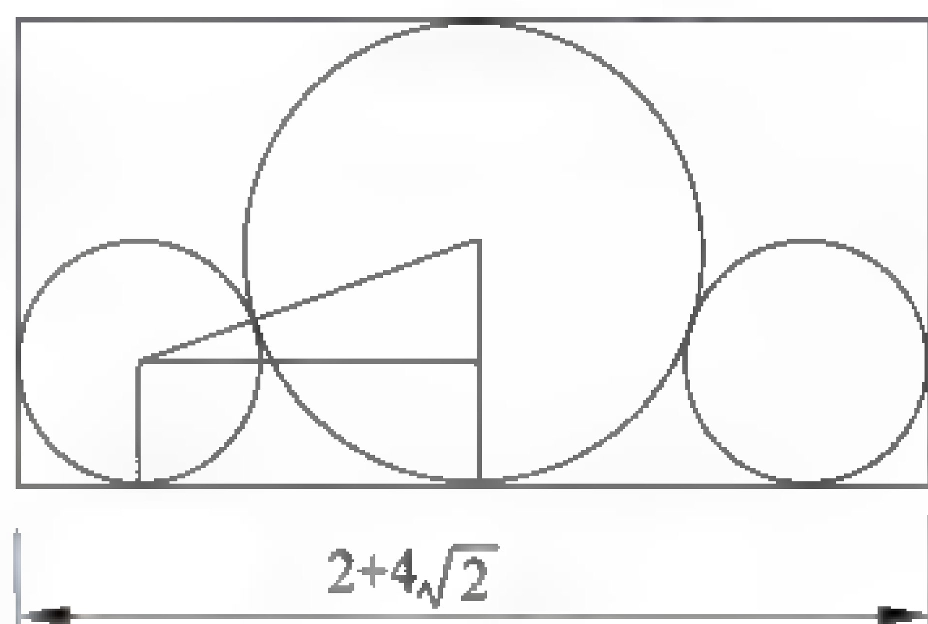


图 6-3　圆排列问题

★ 算法设计

对于给定的 n 个圆，设计一个优先队列式分支限界法，计算 n 个圆的最佳排列方案，使其长度达到最小。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（其中 $1\leq n\leq 20$）。第 2 行有 n 个数，表示 n 个圆的半径。

★ 结果输出

将计算出的最小圆排列的长度输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	7.65685
1 1 2	

★ 评分

如果没有按照题目要求用分支限界法解题，则所得分数减半。

分析与解答：

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    float len;          // 当前圆排列的长度
    int s;              // 待排列圆的个数
    float []x;          // 当前圆排列圆心横坐标
    float []r;          // 当前圆排列

    HeapNode(float ll,int ss, float []xx, float []rr)
    {len=ll;s=ss;x=xx;r=rr;}

    float Center(int t)
    {// 计算当前所选择圆的圆心横坐标
        float temp=0;
```

```
        for(int j=1;j<t;j++){
            float valuex=x[j]+2*(float)Math.sqrt(r[t]*r[j]);
            if(valuex>temp) temp=valuex;
        }
        return temp;
    }
    void Compute(int ii)
    {// 计算当前圆排列的长度
        float low=0,high=0;
        for(int i=1;i<=ii;i++){
            if(x[i]-r[i]<low) low=x[i]-r[i];
            if(x[i]+r[i]>high) high=x[i]+r[i];
        }
        len=high-low;
    }
    public int compareTo(Object x) {
        float   xl=((HeapNode) x).len;
        if (len<xl) return -1;
        if (len==xl) return 0;
        return 1;
    }
}
```

解圆排列问题的优先队列式分支限界法如下：

```
static int n;
static float []p;
static MinHeap heap=new MinHeap();
static float CirclePerm(float []a,int n,float []bestx)
{
    HeapNode E=new HeapNode (Float.MAX_VALUE,0,new float[n+1],a);
    float minlen=Float.MAX_VALUE;
    do{
        if(E.s==n-1){
            E.x[n]=E.Center(n);
            E.Compute(n);
            if(E.len<minlen){
                for(int j=1;j<=n;j++)bestx[j]=E.r[j];
                minlen =E.len;
            }
        }
        else{
            for(int i=E.s+1;i<=n;i++){
              HeapNode N=new HeapNode (E.len,E.s+1,new float[n+1], new float[n+1]);
              for(int j=1;j<=n;j++){N.x[j]=E.x[j];N.r[j]=E.r[j];}
```

```
                N. r[N. s]=E. r[i];N. r[i]=E. r[N. s];
                N. x[N. s]=N. Center(N. s);
                N. Compute(N. s);
                if(N. len<minlen) heap. put(N);
            }
        }
        if(heap. isEmpty()) return minlen;
        E=(HeapNode)heap. removeMin();
    }while(E. len<minlen);
    return minlen;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    n=keyboard. readInt();
    p=new float[n+1];
    float []B=new float[n+1];
    for(int i =1;i<=n;i++) B[i]=keyboard. readFloat();
    System. out. println(CirclePerm(B,n,p));
    for (int i=1; i<=n; i++)   System. out. print(p[i]+" ");
    System. out. println();
}
```

算法实现题 6-9　布线问题

★ 问题描述

假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i,j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r,s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\sum_{1\leqslant i<j\leqslant n} \text{conn}(i,j)\text{dist}(r,s)$。试设计一个优先队列式分支限界法，找出所给 n 个元件的布线成本最小的布线方案。

★ 算法设计

对于给定的 n 个元件，设计一个优先队列式分支限界法，计算最佳布线方案，使布线费用达到最小。

★ 数据输入

由文件 input. txt 给出输入数据。第 1 行有 1 个正整数 n(其中 $1\leqslant n\leqslant 20$)。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数，$1\leqslant i<j\leqslant 20$。

★ 结果输出

将计算出的最小布线费用以及相应的最佳布线方案输出到文件 output. txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

★ **评分**

如果没有按照题目要求用分支限界法解题,则所得分数减半。

分析与解答:

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int s,cd;
    int []x;
    HeapNode(int ss,int cdd, int []xx)
    {s=ss;cd=cdd;x=xx;}
    int len(int [][]conn,int ii)
    {
        int sum=0;
        for(int i=1;i<=ii;i++)
            for(int j=i+1;j<=ii;j++){
                int dist=x[i]>x[j]?x[i]-x[j]:x[j]-x[i];
                sum+=conn[i][j] * dist;
            }
        return sum;
    }
    public int compareTo(Object x) {
        float   xcd=((HeapNode) x).cd;
        if (cd<xcd) return -1;
        if (cd==xcd) return 0;
        return 1;
    }
}
```

解布线问题的优先队列式分支限界法如下:

```
static int n;
static int []p;
static MinHeap heap=new MinHeap();
static int BBArrangeBoards(int [][]conn,int n,int []bestx)
{
    HeapNode E=new HeapNode (0,0,new int[n+1]);
    for(int i=1;i<=n;i++) E.x[i]=i;
    int bestd=Integer.MAX_VALUE;
    while(E.cd<bestd){
```

```
            if(E.s==n-1){
                int ld=E.len(conn,n);
                if(ld<bestd){
                    bestd=ld;
                    for(int j=1;j<=n;j++) bestx[j]=E.x[j];
                }
            }
            else{
                for(int i=E.s+1;i<=n;i++){
                    HeapNode N=new HeapNode (E.s+1,0,new int[n+1]);
                    for(int j=1;j<=n;j++) N.x[j]=E.x[j];
                    N.x[N.s]=E.x[i];
                    N.x[i]=E.x[N.s];
                    N.cd=N.len(conn,N.s);
                    if(N.cd<bestd) heap.put(N);
                }
            }
            if(heap.isEmpty()) return bestd;
            E=(HeapNode)heap.removeMin();
        }
        return bestd;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    p=new int[n+1];
    int [][]B=new int[n+1][n+1];
    for(int i =1;i<=n-1;i++)
        for(int j=i+1;j<=n;j++)B[i][j]=keyboard.readInt();
    System.out.println(BBArrangeBoards(B,n,p));
    for (int i=1; i<=n; i++)   System.out.print(p[i]+" ");
    System.out.println();
}
```

算法实现题 6-10 最佳调度问题

★ 问题描述

假设有 n 个任务由 k 个可并行工作的机器完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

★ 算法设计

对任意给定的整数 n 和 k，以及完成任务 i 需要的时间为 t_i，$i=1\sim n$。设计一个优先队列式分支限界法，计算完成这 n 个任务的最佳调度。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 k。第2行的 n 个正整数是完成 n 个任务需要的时间。

★ 结果输出

将计算出的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	17
2 14 4 16 6 5 3	

★ 评分

如果没有按照题目要求用分支限界法解题，则所得分数减半。

分析与解答：

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int i,dep;
    int []len;
    HeapNode(int ii,int depp, int []ll)
    {i=ii;dep=depp;len=ll;}
    public int compareTo(Object x) {
        float  xl=((HeapNode) x).len[i];
        if (len[i]<xl) return -1;
        if (len[i]==xl) return 0;
        return 1;
    }
}
```

解最佳调度问题的优先队列式分支限界法如下：

```
static int BBMachine()
{
    HeapNode E=new HeapNode (0,0,new int[n+1]);
    for(int i=0;i<k;i++) E.len[i]=0;
    int dep=0;
    while(true){
        if(dep==n){
            int tmp=comp(E.len);
            if(tmp<best)best=tmp;
        }
        else{
            for(int i=0;i<k;i++){
                HeapNode N=new HeapNode (i,dep,new int[n+1]);
                for(int j=0;j<k;j++) N.len[j]=E.len[j];
                N.len[i]+=t[dep];
```

```
                    if(N.len[i]<best) heap.put(N);
                }
            }
            if(heap.isEmpty()) return best;
            E=(HeapNode)heap.removeMin();
            dep=E.dep+1;
        }
    }
```

算法的主函数如下：

```
public static void main(String [] args)
{
    readin();
    System.out.println(BBMachine());
}
```

其中，readin 读入初始数据并作初始化计算。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    k=keyboard.readInt();
    len=new int[k];
    t=new int[n];
    for(int i=0;i<n;i++) t[i]=keyboard.readInt();
    for(int i=0;i<k;i++) len[i]=0;
    best=bound();
}
```

bound 计算上界值。

```
static int bound()
{
    QuickSort.quickSort (t,n);
    for(int i=0;i<n;i++) len[ind(len)]+=t[i];
    return comp(len);
}

static int ind(int []len)
{
    int tmp=0;
    for(int i=1;i<k;i++) if(len[i]<len[tmp])tmp=i;
    return tmp;
}

static int comp(int [ ]len)
{
```

```
    int tmp=0;
    for(int i=0;i<k;i++) if(len[i]>tmp)tmp=len[i];
    return tmp;
}
```

算法实现题 6-11 无优先级运算问题

★ 问题描述

给定 n 个正整数和 4 个运算符 +、-、*、/，且运算符无优先级，如 2+3*5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次，但每种运算符可以任意使用。

★ 算法设计

对于给定的 n 个正整数，设计一个优先队列式分支限界法，用最少的无优先级运算次数产生整数 m。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

★ 结果输出

将计算出的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

★ 评分

如果没有按照题目要求用分支限界法解题，则所得分数减半。

分析与解答：

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int dep;
    int []num;
    int []oper;
    int []flag;
    HeapNode(){
        dep=0;
        num=new int[n];
        oper=new int[n];
        flag=new int[n];
        for(int i=0;i<n;i++) {num[i]=oper[i]=flag[i]=0;}
    }
    public int compareTo(Object x) {
        float  xd=((HeapNode) x).dep;
```

```
            if (dep<xd) return    1;
            if (dep==xd) return 0;
            return 1;
        }
}
```

解无优先级运算问题的优先队列式分支限界法如下：

```
static int m,n,k,dep;
static int []a;
static MinHeap heap=new MinHeap();
static int BBArit()
{
    HeapNode E=new HeapNode();
    while(true){
        if(found(E)){out(E);return 1;}
        else{
            for(int i=0;i<n;i++)
                if (E.flag[i]==0)
                    for(int j=0;j<4;j++){
                        HeapNode N=new HeapNode();
                        newnode(N,E,i,j,dep);
                        heap.put(N);
                    }
        }
        if(heap.isEmpty()) return 0;
        E=(HeapNode)heap.removeMin();
        dep=E.dep+1;
    }
}
```

上述算法中的 found 判定是否找到解；out 输出解；newnode 生成新结点。

```
static boolean found(HeapNode E) {
    int x=E.num[0];
    for(int i=0;i<E.dep;i++){
        switch (E.oper[i]){
            case 0: x+=E.num[i+1]; break;
            case 1: x-=E.num[i+1]; break;
            case 2: x*=E.num[i+1]; break;
            case 3: x/=E.num[i+1]; break;
        }
    }
    return (x==m);
}
static void newnode(HeapNode N,HeapNode E,int i,int j,int dep)
```

```
{
    for(int k=0;k<n;k++){
        N.num[k]=E.num[k];
        N.oper[k]=E.oper[k];
        N.flag[k]=E.flag[k];
    }
    N.dep=dep;
    N.oper[dep]=j;
    N.num[dep]=a[i];
    N.flag[i]=1;
}

static void out(HeapNode E) {
    System.out.println(E.dep);
    for(int i=0;i<E.dep;i++){
        System.out.print(E.num[i]);
        switch (E.oper[i]){
            case 0: System.out.print("+");break;
            case 1: System.out.print("-");break;
            case 2: System.out.print("*");break;
            case 3: System.out.print("/");break;
        }
    }
    System.out.println(E.num[E.dep]);
}
```

算法的主函数如下:

```
public static void main(String [] args)
{
  readin();
  if(BBArit()==0) System.out.println("No Solution!");
}

static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    m=keyboard.readInt();
    a=new int[n];
    for(int i=0;i<n;i++) a[i]=keyboard.readInt();
}
```

算法实现题 6-12　世界名画陈列馆问题

★ 问题描述

世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗,需要在陈列室中设置警卫机器人哨位。每个警卫机器人除了监视它所在的陈列室外,还可以监

视与它所在的陈列室相邻的上、下、左、右4个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人数最少。

★ 算法设计

设计一个优先队列式分支限界法，计算警卫机器人的最佳哨位安排，使得名画陈列馆中每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人数最少。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个正整数 m 和 n，$1 \leqslant m, n \leqslant 20$。

★ 结果输出

将计算出的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第1行是警卫机器人数；接下来的 m 行中每行 n 个数，0表示无哨位，1表示哨位。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

★ 评分

如果没有按照题目要求用分支限界法解题，则所得分数减半。

分析与解答：

堆结点元素类型是 HeapNode。

```
static class HeapNode implements Comparable{
    int i,j,k,t;
    int [][]x;
    int [][]y;

    HeapNode(){
        x=new int[n+2][m+2];
        y=new int[n+2][m+2];
        for(int a=0;a<=n+1;a++)
            for(int b=0;b<=m+1;b++){x[a][b]=0;y[a][b]=0;}
        for(int a=0;a<=m+1;a++){y[0][a]=1;y[n+1][a]=1;}
        for(int a=0;a<=n+1;a++){y[a][0]=1;y[a][m+1]=1;}
        i=1;j=1;k=t=0;
    }

    public int compareTo(Object x) {
        float  xk=((HeapNode) x).k;
        if (k<xk) return  -1;
        if (k==xk) return 0;
        return 1;
```

```
        }
    }
```

解世界名画陈列馆问题的优先队列式分支限界法如下。解空间结点控制关系与算法实现题5-19相同。

```
static int [][]d={{0,0,0},{0,0,0},{0,0,-1},{0,-1,0},{0,0,1},{0,1,0}};
static int n,m,best;
static boolean p;
static int [][]bestx;
static MinHeap heap=new MinHeap();

static void pqbb()
{
    HeapNode E=new HeapNode();
    while(true){
        int i=E.i,j=E.j,k=E.k,t=E.t;
        if(t==n * m){best=k;copy(bestx,E.x);return;}
        else{
            if(i<n)change(E,i+1,j);
            if((j<m)&&((E.y[i][j+1]==0) || (E.y[i][j+2]==0)))change(E,i,j+1);
            if(((E.y[i+1][j]==0)&&(E.y[i][j+1]==0)))change(E,i,j);
        }
        if(heap.isEmpty()) break;
        E=(HeapNode)heap.removeMin();
    }
}

static void change(HeapNode E,int i,int j)
{
    HeapNode N=new HeapNode();
    N.i=E.i;N.j=E.j;N.k=E.k+1;N.t=E.t;
    for(int a=0;a<=n+1;a++)
        for(int b=0;b<=m+1;b++){
            N.x[a][b]=E.x[a][b];
            N.y[a][b]=E.y[a][b];
    }
    N.x[i][j]=1;
    for(int s=1;s<=5;s++){
        int p=i+d[s][1],q=j+d[s][2];
        N.y[p][q]++;
        if(N.y[p][q]==1)N.t++;
    }
    while(!(N.y[N.i][N.j]==0 || N.i>n)){
        N.j++;
        if(N.j>m){N.i++;N.j=1;}
```

```
    }
    heap.put(N);
}
static void copy(int [][]x,int [][]y)
{
    for(int i=0;i<=n;i++)
        for(int j=0;j<=m;j++)x[i][j]=y[i][j];
}
```

compute 完成计算。

```
static void compute()
{
    bestx=new int[n+2][m+2];
    if(n==1 && m==1){System.out.println(1);System.out.println(1);return;}
    pqbb();output();
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
   init();
   compute();
}
static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    m=keyboard.readInt();
    p=false;
    if(n<m){int k=m;m=n;n=k;p=true;}
}
```

算法实现题 6-13　骑士征途问题

★ 问题描述

在一个 $n\times n$ 个方格的国际象棋棋盘上，马（骑士）从任意指定方格出发，按照横 1 步竖 2 步，或横 2 步竖 1 步的跳马规则，走遍棋盘的每一个格子，且每个格子只走 1 次。这样的跳马步骤称为 1 个成功的骑士征途。例如，当 $n=5$ 时的 1 个成功的骑士征途如图 6-4 所示。

	1	2	3	4	5
1	25	14	1	8	19
2	4	9	18	13	2
3	15	24	3	20	7
4	10	5	22	17	12
5	23	16	11	6	21

图 6-4　骑士征途

★ 算法设计

对于给定的 n 和 $n\times n$ 方格的起始位置 x 和 y。用分支限界法找出从指定的方格 (x,y) 出发的一条成

功的骑士征途。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有1个正整数 n, $1 \leqslant n \leqslant 10$;第2行有2个正整数 x 和 y,表示骑士的起始位置为(x,y)。

★ 结果输出

将计算出的成功骑士征途输出到文件 output.txt。如果不存在从(x,y)出发的成功骑士征途,则输出"No solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
5	25 14 1 8 19
1 3	4 9 18 13 2
	15 24 3 20 7
	10 5 22 17 12
	23 16 11 6 21

分析与解答:

与旅行售货员问题类似。

算法实现题 6-14 推箱子问题

★ 问题描述

码头仓库是划分为 $n \times m$ 个格子的矩形阵列。有公共边的格子是相邻格子。当前仓库中有的格子是空闲的;有的格子则已经堆放了沉重的货物。由于堆放的货物很重,单凭仓库管理员的力量是无法移动的。仓库管理员有一项任务,要将一个小箱子推到指定的格子上去。管理员可以在仓库中移动,但不能跨过已经堆放了货物的格子。管理员站在与箱子相对的空闲格子上时,可以做一次推动,把箱子推到另一相邻的空闲格子。推箱时只能向管理员的对面方向推。由于要推动的箱子很重,仓库管理员想尽量减少推箱子的次数。

★ 算法设计

对于给定的仓库布局,以及仓库管理员在仓库中的位置和箱子的开始位置和目标位置,设计一个解推箱子问题的分支限界法,计算出仓库管理员将箱子从开始位置推到目标位置所需的最少推动次数。

★ 数据输入

由文件 input.txt 给出输入数据。输入文件第1行有2个正整数 n 和 m(其中 $1 \leqslant n, m \leqslant 100$),表示仓库是 $n \times m$ 个格子的矩形阵列。接下来有 n 行,每行有 m 个字符,表示格子的状态。

S:表示格子上放了不可移动的沉重货物。

w:表示格子空闲。

M:表示仓库管理员的初始位置。

P:表示箱子的初始位置。

K:表示箱子的目标位置。

★ 结果输出

将计算出的最少推动次数输出到文件 output.txt。如果仓库管理员无法将箱子从开始

位置推到目标位置，则输出“No solution!”。

输入文件示例	输出文件示例
input. txt	output. txt
10 12	7
SSSSSSSSSSSS	
SwwwwwwwSSSS	
SwSSSSwwSSSS	
SwSSSSwwSKSS	
SwSSSSwwSwSS	
SwwwwwPwwwww	
SSSSSSSwSwSw	
SSSSSSMwSwww	
SSSSSSSSSSSS	
SSSSSSSSSSSS	

分析与解答：

与布线问题类似，结点元素类型是 position。

```
static class position {
    int row,col,dir;
    position(){}
    position(int r,int c,int d) {row=r;col=c;dir=d;}
}
```

它的成员 row 和 col 分别表示方格所在的行和列；dir 表示推的方向。

算法用到如下全局变量：

```
static int []op={1,0,3,2};
static int m,n,totm,markr;
static int [][]grid;
static int [][]reach;
static int [][]mark;
static int [][]low;
static int [][]totr;
static int [][][]comp;
static long [][][]ans;
static position start,finish,man;
static position []offset=new position [4];
```

init 实现数据输入及预处理。

```
static void init()
{
    char c;
    ReadStream keyboard=new ReadStream();
```

```
        n=keyboard.readInt();
        m=keyboard.readInt();
        grid=new int[n+2][m+2];
        reach=new int[n+1][m+1];
        mark=new int[n+1][m+1];
        low=new int[n+1][m+1];
        totr=new int[n+1][m+1];
        ans=new long[n+1][m+1][4];
        comp=new int[n+1][m+1][10];
        for(int i=0;i<n+2;i++)
            for(int j=0;j<m+2;j++)grid[i][j]=0;
        start=new position();
        finish=new position();
        man=new position();
        for(int i=0;i<4;i++) offset[i]=new position();
        for(int i=1;i<=n;i++)
            for(int j=1;j<=m;j++){
                while(true) {
                    c=keyboard.readChar();
                    if(Character.isLetter(c))break;
                }
                if(c=='M') {man.row=i;man.col=j;}
                if(c=='P') {start.row=i;start.col=j;}
                if(c=='K') {finish.row=i;finish.col=j;}
                if(c=='S') grid[i][j]=1;
            }
        // 设置方格阵列"围墙"
        for(int i=0;i<=m+1;i++)
            grid[0][i]=grid[n+1][i]=1;        // 顶部和底部
        for(int i=0;i<=n+1;i++)
            grid[i][0]=grid[i][m+1]=1;        // 左翼和右翼
        // 初始化相对位移
        offset[0].row=0;offset[0].col=-1;     // 左
        offset[1].row=0;offset[1].col=1;      // 右
        offset[2].row=-1;offset[2].col=0;     // 上
        offset[3].row=1;offset[3].col=0;      // 下
        prepro();
        for(int i=0;i<=n;i++)
            for(int j=0;j<=m;j++){
                reach[i][j]=0;
                for(int k=0;k<4;k++)ans[i][j][k]=Long.MAX_VALUE;
            }
        dfs(man.row,man.col);
    }
```

其中,prepro 对方格连通性作预处理计算。

```
static void prepro()
{
    totm=0;markr=0;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++){
            mark[i][j]=-1;
            low[i][j]=Integer.MAX_VALUE;
            totr[i][j]=0;
            reach[i][j]=-1;
        }
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            if(grid[i][j]==0 && mark[i][j]==-1)fill(i,j);
}

static void put(int x,int y,int a,int b)
{
    if(x==a && y==b) return;
    if(reach[x][y]==2) return;
    comp[x][y][totr[x][y]]=markr;
    totr[x][y]++;
    reach[x][y]=2;
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row,y1=y+offset[i].col;
        if(grid[x1][y1]==0) put(x1,y1,a,b);
    }
}

static void fill(int x,int y)
{
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row,y1=y+offset[i].col;
        if(grid[x1][y1]==0){
            if(mark[x1][y1]==-1){
                mark[x1][y1]=totm;totm++;
                fill(x1,y1);
                low[x][y]=Math.min(low[x][y],low[x1][y1]);
                if(low[x1][y1]>=mark[x][y]){
                    markr++;
                    put(x1,y1,x,y);
                    comp[x][y][totr[x][y]]=markr;
                    totr[x][y]++;
                    reach[x][y]=1;
                }
            }
            else low[x][y]=Math.min(low[x][y],mark[x1][y1]);
```

```
            }
        }
    }
```

预处理后,由 connect 计算方格连通性。

```
static boolean connect(int x1,int y1,int x2,int y2)
{
    for(int i=0;i<totr[x1][y1];i++)
        for(int j=0;j<totr[x2][y2];j++)
            if(comp[x1][y1][i]==comp[x2][y2][j]) return true;
    return false;
}
```

dfs 计算初始可达性。

```
static void dfs(int x,int y)
{
    if(reach[x][y]==1) return;
    reach[x][y]=1;
    for(int i=0;i<4;i++){
        int x1=x+offset[i].row,y1=y+offset[i].col;
        if(grid[x1][y1]==0 &&(x1!=start.row || y1!=start.col)) dfs(x1,y1);
    }
}
```

解推箱子问题的队列式分支限界法 push 如下:

```
static void push()
{
    ArrayQueue queue=new ArrayQueue();
    position here,nbr;
    for(int i=0;i<4;i++){
        nbr=new position (start.row,start.col,i);
        if(ok(start.row+offset[i].row,start.col+offset[i].col)){
            ans[start.row][start.col][i]=0;
            queue.put(nbr);
        }
    }
    while(!queue.isEmpty()){
        here=(position)queue.remove();
        int d=here.dir,x=here.row+offset[op[d]].row,y=here.col+offset[op[d]].col;
        if(grid[x][y]==0 && ans[x][y][d]>ans[here.row][here.col][d]+1){
            ans[x][y][d]=ans[here.row][here.col][d]+1;
            nbr=new position (x,y,d);
            queue.put(nbr);
            for(int i=0;i<4;i++)
                if(i!=d){
```

```
                    int x1=x+offset[i].row,y1=y+offset[i].col;
                    if(grid[x1][y1]==0 && connect(x1,y1,here.row,here.col) &&
                        ans[x][y][i]>ans[x][y][d]){
                        ans[x][y][i]=ans[x][y][d];
                        nbr=new position (x,y,i);
                        queue.put(nbr);
                    }
                }
            }
        }
}
static boolean ok(int a,int b)
{
    return grid[a][b]==0 && reach[a][b]==1;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
   init();
   push();
   out();
}
```

out 输出计算结果。

```
static void out()
{
    long min=ans[finish.row][finish.col][0];
    for(int i=1;i<4;i++)
        if(ans[finish.row][finish.col][i]<min) min=ans[finish.row][finish.col][i];
    if(min==Long.MAX_VALUE) System.out.println("No solution!");
    else System.out.println(min);
}
```

算法实现题 6-15　图形变换问题

★ 问题描述

给定两个 4×4 方格阵列组成的图形 A(见图 6-5(a))和 B(见图 6-5(b)),每个方格的颜色为黑色或白色,如图 6-5 所示。方格阵列中有公共边的方格称为相邻方格。图形变换问题的每一步变换可以交换相邻方格的颜色。试设计一个算法,计算最少需要多少步变换,才能将图形 A 变换为图形 B。

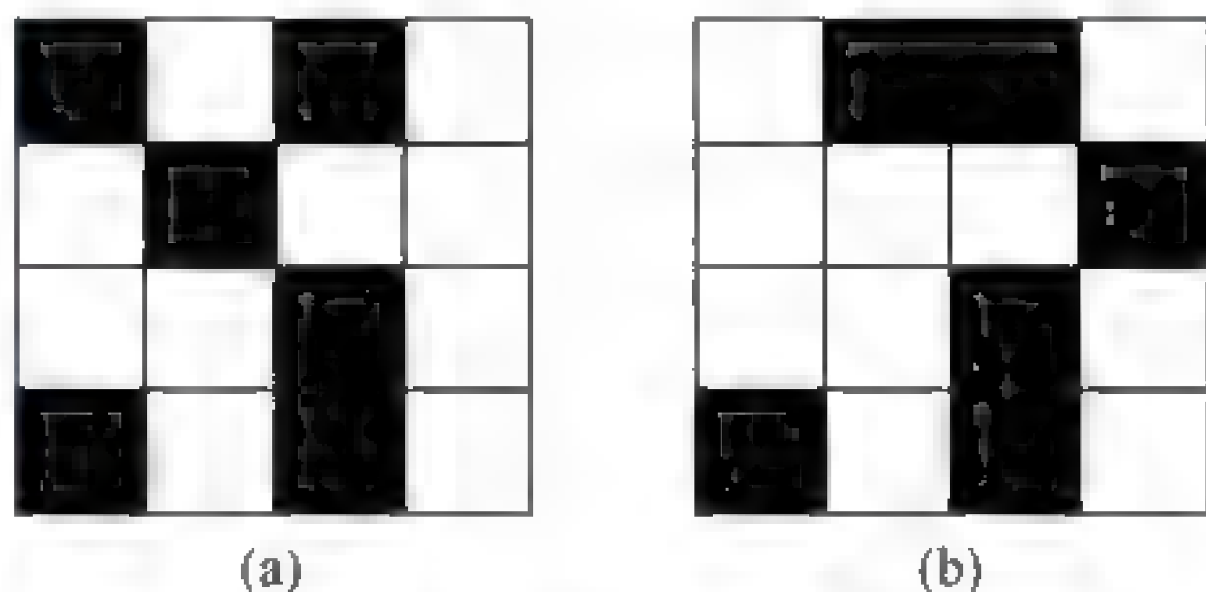


图 6-5　图形变换问题

★ 算法设计

对于给定的两个方格阵列,计算将图形 A 变

换为图形B的最少变换次数。

★ **数据输入**

由文件input.txt给出输入数据。前4行是图形A的方格阵列,后4行是图形B的方格阵列。0表示白色,1表示黑色。

★ **结果输出**

将计算出的最少变换次数和相应的变换序列输出到文件output.txt。第1行是最少变换次数。从第2行开始,每行用4个数表示一次变换。例如,1112表示交换方格(1,1)和(1,2)的颜色。问题无解时,则输出"No solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
1010	3
0100	1112
0010	2223
1010	2324
0110	
0001	
0010	
1010	

分析与解答:

由于图形由16个方格组成,可以用16位的整型数表示图形方格的颜色状态。

解图形变换问题的队列式分支限界法fifobb如下:

```
static final int MaxSize=1<<16;
static int sour,dest;
static int []a=new int[MaxSize];
static int []b=new int[MaxSize];

static boolean fifobb()
{
    ArrayQueue queue=new ArrayQueue();
    int E=0;
    for(int i=1;i<MaxSize;i++)a[i]=-1;
    a[sour]=0;
    queue.put(new Integer(sour));
    while(!queue.isEmpty()){
        E=((Integer)queue.remove()).intValue();
        for(int j=0,mode=3;j<16;j++,mode<<=1)
            if(j%4!=3 &&(((E & mode)>>j)==1 || ((E & mode)>>j)==2)){
                int N=E^mode;
                if(a[N]==-1){a[N]=a[E]+1;b[N]=j;queue.put(new Integer(N));}
            }
        for(int j=0,mode=17;j<12;j++,mode<<=1)
```

```
            if(((E & mode)>>j)==1 || ((E & mode)>>j)==16){
                int N=E^mode;
                if(a[N]==-1){a[N]=a[E]+1;b[N]=-j-1;queue.put(new Integer(N));}
            }
        if(a[dest]!=-1) return true;
    }
    return false;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
  init();
  if(fifobb()){System.out.println(a[dest]);output(dest);}
  else System.out.println("No solution!");
}

static void init()
{
    int i;char c;
    ReadStream keyboard=new ReadStream();
    for(sour=i=0;i<16;i++){
        c=keyboard.readChar();
        if(!Character.isDigit(c))c=keyboard.readChar();
        sour|=(int)(c-'0')<<i;
    }
    for(dest=i=0;i<16;i++){
        c=keyboard.readChar();
        if(!Character.isDigit(c))c=keyboard.readChar();
        dest|=(int)(c-'0')<<i;
    }
}

static void output(int E)
{
    if(E==sour) return;
    int last=E;
    if(b[E]>=0) last^=3<<b[E];
    else last^=17<<(-b[E]-1);
    output(last);
    if(b[E]>=0)
        System.out.println((b[E]/4+1)+" "+(b[E]%4+1)+" "+(b[E]/4+1)+" "
                          +(b[E]%4+2));
    else
        System.out.println(((-b[E]-1)/4+1)+" "+((-b[E]-1)%4+1)+" "
                          +((-b[E]-1)/4+2)+" "+((-b[E]-1)%4+1));
}
```

算法实现题 6-16　行列变换问题

★ 问题描述

给定两个 $m \times n$ 方格阵列组成的图形 A(见图 6-6(a))和 B(见图 6-6(b)),每个方格的颜色为黑色或白色,如图 6-6 所示。行列变换问题的每一步变换可以交换任意 2 行或 2 列方格的颜色,或者将某行或某列颠倒。上述每次变换算作一步。试设计一个算法,计算最少需要多少步,才能将图形 A 变换为图形 B。

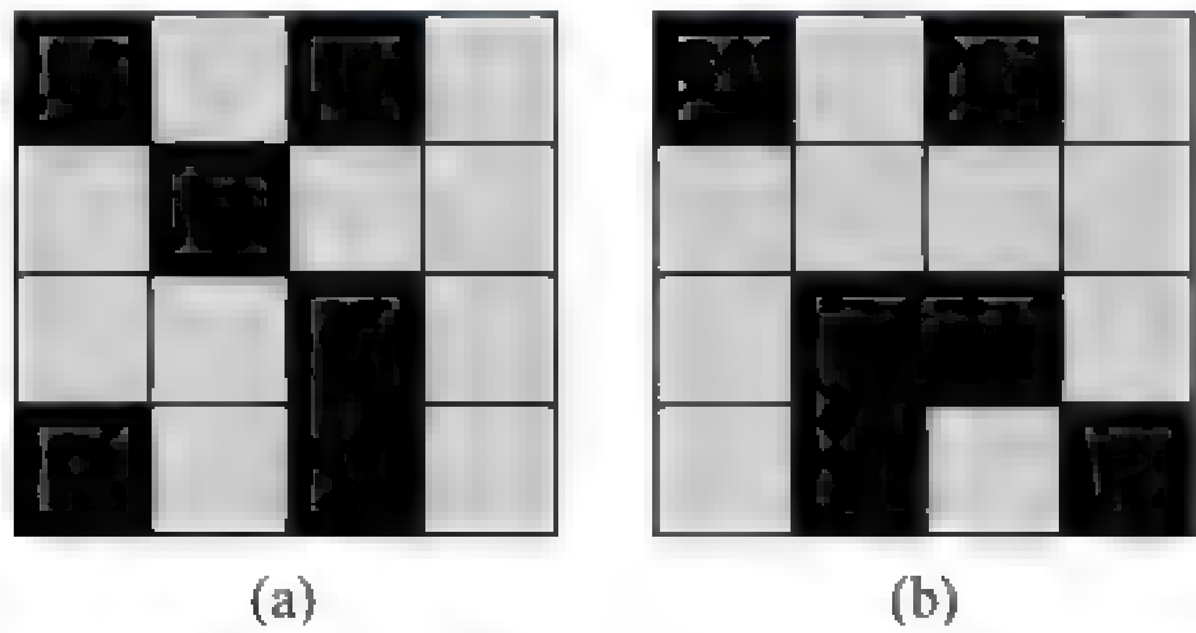


图 6-6　行列变换问题

★ 算法设计

对于给定的两个方格阵列,计算将图形 A 变换为图形 B 的最少变换次数。

★ 数据输入

由文件 input.txt 给出输入数据。文件的第 1 行有 2 个正整数 m 和 n。以下的 m 行是方格阵列的初始状态 A,每行有 n 个数字表示该行方格的状态,0 表示白色,1 表示黑色。接着的 m 行是方格阵列的目标状态 B。

★ 结果输出

将计算出的最少变换次数和相应的变换序列输出到文件 output.txt。第 1 行是最少变换次数。从第 2 行开始,依次输出变换的图形序列。问题无解时,则输出"No solution!"。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	2
1010	
0100	1010
0010	0100
1010	0010
1010	1010
0000	
0110	1010
0101	0000
	0110
	1010
	1010
	0000
	0110
	0101

分析与解答:

与图形变换问题类似。

算法实现题 6-17　重排 n^2 宫问题

★ 问题描述

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事“关羽义释曹操”而设计的智力玩具“华容道”，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1～8 按照任意次序排在 3×3 的方格阵列中，留下一个空格。与空格相邻的数字，允许从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1～8 按行排好序。在一般情况下，重排 n^2 宫问题是将数字 $1 \sim n^2-1$ 按照任意次序排在 $n \times n$ 的方格阵列中，留下一个空格。允许与空格相邻的数字从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将初始状态变换到目标状态，如图 6-7 所示。

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

图 6-7　重排 n^2 宫问题

★ 算法设计

对于给定的 $n \times n$ 方格阵列中数字 $1 \sim n^2-1$ 初始排列和目标状态，计算将初始排列通过合法移动变换为目标状态最少移动次数。

★ 数据输入

由文件 input.txt 给出输入数据。文件的第 1 行有 1 个正整数 n。以下的 n 行是 $n \times n$ 方格阵列中的数字 $1 \sim n^2-1$ 的初始排列，每行有 n 个数字表示该行方格中的数字，0 表示空格。接着的 n 行是方格阵列中数字 $1 \sim n^2-1$ 的目标状态。

★ 结果输出

将计算出的最少移动次数和相应的移动序列输出到文件 output.txt。第 1 行是最少移动次数。从第 2 行开始，依次输出移动序列。用大写英文字母 D，U，L，R 分别表示向下、向上、向左、向右移动。问题无解时，则输出“No solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
3	2
1 2 3	DR
4 0 6	
7 5 8	
1 2 3	
4 5 6	
7 8 0	

分析与解答：

解重排 n^2 宫问题的优先队列式分支限界法用当前状态到目标状态的 manhattan 距离为结点的优先级。这是一种启发式的搜索策略。由于 manhattan 距离是单调的，所以按此优先级搜索的优先队列式分支限界算法是一个 A^* 算法，从而保证找到的解是移动次数最少的解。

逐步深化的 A^* 搜索算法（IDA*）与上述算法相比，只用很少的内存，时间效率也相当

高,具体算法描述如下:

用类 Board 表示 $n \times n$ 方格阵列。

```
static class Board{
    int y,x,dist,psz;
    int []boardm;
    int []path;
    Board(Board bd)
    {
        x=bd.x;y=bd.y;dist=bd.dist;psz=bd.psz;
        path=new int[plen];
        boardm=new int[boardsz];
        for(int j=0;j<plen;j++)path[j]=bd.path[j];
        for(int i=0;i<boardsz;i++)boardm[i]=bd.boardm[i];
    }
    Board()
    {
        psz=0;dist=0;
        path=new int[plen];
        boardm=new int[boardsz];
    }
    int heur(){return dist;}
    void getboard(int []m)
    {
        for(int i=0;i<boardsz;i++){
            boardm[i]=m[i];
            if(m[i]==0){y=i/rowsz;x=i%rowsz;}
        }
        for(int i=0;i<boardsz;i++)
            if(boardm[i]!=0)dist+=getdist(boardm[i],i);
    }
    // 按3个方向移动
    boolean move(int dir)
    {
        int nx=x+step[dir][0],ny=y+step[dir][1];
        if((psz==0 || op[dir]!=path[psz-1]) &&
            nx>-1 && nx<rowsz && ny>-1 && ny<rowsz){
            dist=dist+1-getdist(boardm[ny*rowsz+nx],ny*rowsz+nx)
                +getdist(boardm[ny*rowsz+nx],y*rowsz+x);
            MyMath.swap(boardm,y*rowsz+x,ny*rowsz+nx);
            y=ny;x=nx;
            path[psz++]=dir;
            return true;
        }
```

```
        else return false;
    }
    // 计算 manhattan 距离
    int getdist(int v,int loc)
    {
        int dis=Math.abs((pos[v]%rowsz)-(loc%rowsz));
        dis+=Math.abs((pos[v]/rowsz)-(loc/rowsz));
        return dis;
    }
    // 到达目标状态
    boolean reached()
    {
        for(int i=0;i<boardsz;i++)
            if(boardm[i]!=dest[i])return false;
        return true;
    }
    void out()
    {
        char []dir={'U','D','L','R'};
        System.out.println(psz);
        for(int i=0;i<psz;i++){
            System.out.print(dir[path[i]]);
            if(i%20==19) System.out.println();
        }
        if(psz%20!=0)System.out.println();
    }
}
```

逐步深化的 A^* 搜索算法 idastar 描述如下：

```
static boolean solve(int dep,Board E)
{
    if(dep+E.dist<=maxdep){
        if(E.reached()){E.out();return true;}
        for(int i=0;i<4;i++){
            Board N=new Board (E);
            if(N.move(i))
                if(solve(dep+1,N))return true;
        }
    }
    return false;
}
// IDA* 算法
static void idastar()
{
```

```
        Board E=new Board();
        E.getboard(sour);
        maxdep=E.heur();
        if(maxdep==0) {System.out.println(0);return;}
        while(!solve(0,E)) maxdep+=2;
    }
```

算法的主函数如下：

```
static final int plen=60;
static int rowsz,boardsz,maxdep;
static int []op={1,0,3,2};
static int [][]step={{0,-1},{0,1},{-1,0},{1,0}};
static int []sour;
static int []dest;
static int []pos;

public static void main(String [] args)
{
   init();
   if(odd())idastar();
   else System.out.println("No Solution!");
}
```

init 读入初始数据。

```
static void init()
{
      ReadStream keyboard=new ReadStream();
      rowsz=keyboard.readInt();
      boardsz=rowsz * rowsz;
      sour=new int[boardsz];
      dest=new int[boardsz];
      pos=new int[boardsz];
      for(int i=0;i<boardsz;i++)sour[i]=keyboard.readInt();
      for(int i=0;i<boardsz;i++){dest[i]=keyboard.readInt();pos[dest[i]]=i;}
}
```

$n\times n$ 方格阵列中的数字按照从上到下、从左到右的顺序排列，对应于数字 $0\sim n^2-1$ 的一个排列。设 $m=n^2-1$，初始排列为 $(s_0,s_1,\cdots,s_m)$，目标排列为 $(t_0,t_1,\cdots,t_m)$。初始排列中空格位置为 (x_0,y_0)；目标排列中空格位置为 (x_1,y_1)。(x_0,y_0) 与 (x_1,y_1) 的 manhattan 距离为 $\text{dist}_0=|x_1-x_0|+|y_1-y_0|$。初始排列 $(s_0,s_1,\cdots,s_m)$ 到目标排列 $(t_0,t_1,\cdots,t_m)$ 的变换对应于置换 $\begin{pmatrix} s_0,s_1,\cdots,s_m \\ t_0,t_1,\cdots,t_m \end{pmatrix}$。该置换的奇偶性与 dist_0 的奇偶性相同时问题有解，否则问题无解。由此可见，对于给定的初始排列，只有 $(n^2)!/2$ 个排列是从初始排列可到达的。

下面的算法 odd 计算初始状态到目标状态的置换奇偶性，由此判定问题的可解性。

```
static boolean odd()
{
    int count=0,count1=0,i1=0,j1=0,i2=0,j2=0;
    int []c=new int[boardsz];
    for(int k=0;k<boardsz;k++){
        c[dest[k]]=sour[k];
        if(sour[k]==0){i1=k/rowsz+1;j1=k%rowsz+1;}
        if(dest[k]==0){i2=k/rowsz+1;j2=k%rowsz+1;}
    }
    int posi=((i1+i2)%2+(j1+j2)%2)%2;
    for(int j=0;j<boardsz;j++){
        int k=c[j],k1=j;
        count1=0;
        while(k>=0){k=c[k1];c[k1]=-1;k1=k;count1++;}
        if(count1>0) count+=count1-2;
    }
    if(count%2==posi)return true;
    else return false;
}
```

算法实现题 6-18　最长距离问题

★ 问题描述

重排九宫是一个古老的单人智力游戏。据说重排九宫起源于我国古时由三国演义故事"关羽义释曹操"而设计的智力玩具"华容道"，后来流传到欧洲，将人物变成数字。原始的重排九宫问题是这样的：将数字 1～8 按照任意次序排在 3×3 的方格阵列中，留下一个空格。与空格相邻的数字，允许从上、下、左、右 4 个方向移动到空格中。游戏的最终目标是通过合法移动，将数字 1～8 按行排好序。最长距离问题考查的是，从数字 1～8 在 3×3 的方格阵列的初始排列 A(见图 6-8(a))出发，找出与其相应的最长距离目标状态 B(见图 6-8(b))。换句话说，从 A 到 B 的最优移动序列的长度最长，如图 6-8 所示。

(a)

1	2	3
4		6
7	5	8

(b)

1	2	3
4	5	6
7	8	

图 6-8　最长距离问题

★ 算法设计

对于给定的 3×3 方格阵列中数字 1～8 初始排列，计算与初始排列相应的最长距离目标状态。

★ 数据输入

由文件 input.txt 给出输入数据。文件有 3 行，每行有 3 个数字表示该行方格中的数字，0 表示空格。

★ 结果输出

将计算出的最长距离目标状态输出到文件 output.txt。第 1 行有 2 个正整数 x 和 y，x 是最长距离的值，y 是最长距离目标状态个数。从第 2 行开始，依次输出最长距离目标状态和到达该最长距离目标状态的最优移动序列。用大写英文字母 D，U，L，R 分别表示向

下、向上、向左、向右移动。

输入文件示例

input.txt

```
2 6 4
1 3 7
0 5 8
```

输出文件示例

output.txt

```
31 2
8 7 1
0 3 5
4 6 2
UURDDLUURRDDLURDLLUURDLURRDDLLU
8 1 5
7 3 6
4 0 2
UURDDRULLURRDLLDRRULULDDRUULDDR
```

分析与解答:

算法中用到的一些变量如下:

```
static final int Bitsint=32;
static final int Rowsz=3;
static final int Permsz=Rowsz * Rowsz;
static char []op={'U','D','L','R'};
static class Node {
    int code,dep,dir,parent;
}
static Node []buff;
static int first,last,fact,masksz;
static int []bitset;
static int []sour=new int[Permsz];
static int []perm=new int[Permsz];
```

由于总共有 9!/2=181 440 个可达状态。可用数组 buff 记录所有状态。每个状态对应于数字 0～8 的一个排列,对每个排列编序。

```
static int ptoi(int []perm)
{
    int []pcpy=new int[Permsz];
    int []invp=new int[Permsz];
    int n=0;
    for(int i=0;i<Permsz;++i){
        pcpy[i]=perm[i];
        invp[perm[i]]=i;
    }
    for(int i=Permsz-1;i>0;--i){
        int p=invp[i];
```

```
            pcpy[p]=pcpy[i];pcpy[i]=i;
            invp[pcpy[p]]=p;invp[i]=i;
            n*=i+1;n+=i-p;
        }
        return n;
}
```

反之,给定 0～362 879 中的一个数,可唯一确定数字 0～8 的一个排列。

```
static void itop(int perm[],int n)
{
    perm[0]=0;
    for(int i=1;i<Permsz;++i){
        int p=i-n%(i+1);
        perm[i]=perm[p];perm[p]=i;n/=i+1;
    }
}
```

用下面的队列式分支限界法 fifobb 解此问题。

```
static void fifobb()
{
    while(first<fact/2){
        int code=buff[first].code;
        itop(perm,code);
        int i;
        for(i=0;perm[i]!=Permsz-1&&i<Permsz;++i);
        int brow=i/Rowsz;
        int bcol=i%Rowsz;
        for(int d=0;d<4;d++) trymove(brow,bcol,i,d);
        first++;
    }
}
```

trymove 按照可移动方向扩展结点。

```
static void trymove(int brow,int bcol,int i,int d)
{
    switch(d){
        case 0:
            if(brow>0){
                MyMath.swap(perm,i,i-Rowsz);
                addperm(d);
                MyMath.swap(perm,i,i-Rowsz);
            }
            break;
        case 1:
```

```
                if(brow<Rowsz-1){
                    MyMath.swap(perm,i,i+Rowsz);
                    addperm(d);
                    MyMath.swap(perm,i,i+Rowsz);
                }
                break;
            case 2:
                if(bcol>0){
                    MyMath.swap(perm,i,i-1);
                    addperm(d);
                    MyMath.swap(perm,i,i-1);
                }
                break;
            case 3:
                if(bcol<Rowsz-1){
                    MyMath.swap(perm,i,i+1);
                    addperm(d);
                    MyMath.swap(perm,i,i+1);
                }
        }
}
```

Addperm 将新结点存储到数组 buff 中。

```
static void addperm(int dir)
{
    int code=ptoi(perm);
    int m=code/Bitsint;
    int n=code%Bitsint;
    if(((bitset[m]>>n)&1)>0) return;
    bitset[m]|=1<<n;
    buff[last].parent=first;
    buff[last].dir=dir;
    buff[last].dep=buff[first].dep+1;
    buff[last++].code=code;
}
```

算法的主函数如下：

```
public static void main(String [] args)
{
   init();
   fifobb();
   output();
}
```

init 读入初始数据并作初始化计算。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    fact=1;
    for(int i=0,n=0;i<Permsz; ++i){
        n=keyboard.readInt();
        if(n==0)sour[i]=Permsz-1;
        else sour[i]=n-1;
        fact *=i+1;
        perm[i]=sour[i];
    }
    int icode=ptoi(sour);
    masksz=(fact+Bitsint-1)/Bitsint;
    bitset=new int[masksz];
    for(int i=0;i<masksz;++i) bitset[i]=0;
    bitset[icode/Bitsint]|=1<<(icode%Bitsint);
    first=0;last=0;
    buff=new Node[fact/2];
    for(int i=0;i<fact/2;i++)buff[i]=new Node();
    buff[last].parent=-1;
    buff[last].dep=0;
    buff[last++].code=icode;
}
```

output 输出计算结果。

```
static void output()
{
    int i,j,best=0;
    for(i=0,j=0;i<fact/2;++i)
        if(buff[i].dep>=best){best=buff[i].dep;j=i;}
    for(i=0,j=0;i<fact/2;++i)
        if(buff[i].dep==best)j++;
    System.out.println(best+" "+j);
    for(i=0,j=0;i<fact/2;++i)
        if(buff[i].dep==best){
            itop(perm,buff[i].code);
            outperm();outmove(i);System.out.println();
        }
}
static void outmove(int first)
{
    if(buff[first].dep==0)return;
```

```
        outmove(buff[first].parent);
        System.out.print(op[buff[first].dir]);
    }
    static void outperm()
    {
        for(int i=0;i<Permsz;i++){
            System.out.print((perm[i]+1)%Permsz+" ");
            if((i+1)%Rowsz==0)System.out.println();
        }
    }
```

第 7 章 概率算法

习题 7-1　模拟正态分布随机变量

在实际应用中，常需模拟服从正态分布的随机变量，其密度函数为 $\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-a)^2}{2\sigma^2}}$，其中，$a$ 为均值，σ 为标准差。

如果 s 和 t 是$(-1,1)$中均匀分布的随机变量，且 $s^2+t^2<1$，令 $p=s^2+t^2$，$q=\sqrt{(-2\ln p)/p}$，$u=sq$，$v=tq$，则 u 和 v 是服从标准正态分布($a=0$，$\sigma=1$)的两个互相独立的随机变量。

(1) 利用上述事实，设计一个模拟标准正态分布随机变量的算法。

(2) 将上述算法扩展到一般的正态分布。

分析与解答：

(1) 模拟标准正态分布随机变量的算法如下：

```
public double Norm()
{
    double s,t,p,q;
    while(true){
        s=2 * fRandom()-1;
        t=2 * fRandom()-1;
        p=s * s+t * t;
        if(p<1)break;
    }
    q=Math.sqrt((-2 * Math.log(p))/p);
    return s * q;
```

(2) 扩展到一般的正态分布的算法如下：

```
public double Norm(double a,double b)
{
    double x=Norm();
    return a+b * x;
}
```

习题 7-2　随机抽样算法

设有一个文件含有 n 个记录。

(1) 试设计一个算法随机抽取该文件中 m 个记录。

(2) 如果事先不知道文件中记录个数,应如何随机抽取其中的 m 个记录。

分析与解答:

(1) 以概率 m/n 抽取记录。该方法的标准差是 $\sqrt{m(1-m/n)}$,有时可能不满足要求。假设在前 t 次考查的记录中,已抽取了 k 个记录,接下来第 $t+1$ 次考查取得第 $k+1$ 个记录的概率为:$\binom{n-t-1}{n-k-1}\Big/\binom{n-t}{n-k}=\frac{n-k}{n-t}$。按此概率抽取记录是无偏的。

抽样算法如下:

```
static void samp(int n,int m,int s[])
{
    Random rnd=new Random();
    int x=0,y=0,k;
    while(y<m){
        double u=rnd.fRandom();
        k=(int)(u*n);
        if(u*(n-x)<(n-y)){s[k]++;y++;}
        x++;
    }
}
```

(2) 如果事先不知道文件中记录个数,通常可以先进行一次扫描,确定记录个数后再抽样。另一个较好的方法是在扫描时预先随机抽取 $p>m$ 个记录,然后对抽取出的 p 个记录作两次抽样,从中随机抽取 m 个记录。

习题 7-3　随机产生 m 个整数

试设计一个算法随机地产生范围在 $1\sim n$ 中的 m 个随机整数,且要求这 m 个随机整数互不相同。

分析与解答:

与习题 7-2 类似,解此问题的 floyd 算法如下:

```
static void floyd(int n,int m,int s[])
{
    Random rnd=new Random();
    int y=0;
    while(y<m){
        for(int j=n-m+1;j<=n;j++){
            double u=rnd.fRandom();
            int k=(int)(1+j*u);
            System.out.println(k);
            if(s[k]==0){s[k]++;y++;}
            else if(s[j]==0){s[j]++;y++;}
```

```
        }
    }
```

习题 7-4　集合大小的概率算法

设 X 是含有 n 个元素的集合，从 X 中均匀地选取元素。设第 k 次选取时首次出现重复。

(1) 试证明当 n 充分大时，k 的期望值为 $\beta\sqrt{n}$，其中，$\beta=\sqrt{\frac{\pi}{2}}=1.253$。

(2) 由此设计一个计算给定集合 X 中元素个数的概率算法。

分析与解答：

(1) 从含有 n 个元素的集合 X 中均匀地选取元素，第 k 次选取时首次出现重复的概率为

$$\frac{\binom{n}{k-1}(k-1)!(k-1)}{n^k}$$

k 的期望值为

$$E(k)=\sum_{k=1}^{n}\frac{\binom{n}{k-1}(k-1)!(k-1)k}{n^k}$$

用 Stirling 公式

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+\Theta\left(\frac{1}{n^2}\right)\right)$$

代入计算可得

$$E(k)=\sqrt{\frac{\pi n}{2}}-\frac{1}{3}+\Theta\left(\frac{1}{\sqrt{n}}\right)$$

(2) 由(1)的结果可设计计算给定集合中元素个数的概率算法如下：

```
int count(Set x)
{
    int k=0;
    Set S=new Set();
    int a=uniform(x);
    while(true){
        S.insert(a);
        k++;a=uniform(x);
        if(S.find(a))break;
    }
    return (int)(2 * k * k/pi);
}
```

习题 7-5　生日问题

试设计一个近似算法计算 $365!/340!365^{25}$，并精确到 4 位有效数字。

分析与解答：

该问题中的数是概率论中著名的生日问题的解答。在 k 个人中，至少 2 人有相同生日

的概率为：$1-365!/(365-k)!365^k$。

在一般情况下，$n!/(n-k)!n^k$ 可以用 Stirling 公式化简后作近似计算。

由 Stirling 公式，

$$n! = \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

和

$$\ln(x) = x - x^2/2 + \Theta(x^3)$$

$$n!/(n-k)!n^k = \left(1+\Theta\left(\frac{1}{n}\right)\right)\left(\frac{\sqrt{2\pi n}}{\sqrt{2\pi(n-k)}}\right)\frac{\left(\frac{n}{\mathrm{e}}\right)^n}{\left(\frac{n-k}{\mathrm{e}}\right)^{n-k}n^k}$$

$$=\left(1+\Theta\left(\frac{1}{n}\right)\right)\left(\frac{n}{n-k}\right)^{n-k}\mathrm{e}^{-k} = \left(1+\Theta\left(\frac{1}{n}\right)\right)\mathrm{e}^{(n-k)\ln\left(1+\frac{k}{n-k}\right)-k}$$

$$=\left(1+\Theta\left(\frac{1}{n}\right)\right)\mathrm{e}^{(n-k)\left(1+\frac{k}{n-k}-\frac{k^2}{2(n-k)^2}+\Theta\left(\frac{1}{(n-k)^3}\right)\right)-k}$$

$$=\left(1+\Theta\left(\frac{1}{n}\right)\right)\mathrm{e}^{-k^2/2n+\Theta\left(\frac{1}{n^2}\right)}$$

由此可得 $n!/(n-k)!n^k \approx \mathrm{e}^{-k^2/2n}$。

用更精确些的估计式

$$n! = \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n\left(1+\frac{1}{12n}+\Theta\left(\frac{1}{n^2}\right)\right)$$

和

$$\ln(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \Theta(x^4)$$

可得 $n!/(n-k)!n^k = \mathrm{e}^{\frac{-k(k-1)}{2n-\frac{k^3}{6n^2}}\pm O\left(\max\left(\frac{k^2}{n^2},\frac{k^4}{n^3}\right)\right)}$。

由此可以求得：$365!/340!365^{25} = 0.4311$；$1-365!/340!365^{25} = 0.5689$。

习题 7-6 易验证问题的拉斯维加斯算法

一个问题是易验证的是指对该问题的给定实例的每一个解，都可以有效地验证其正确性。例如求一个整数的非平凡因子问题是易验证的，而求一个整数的最小非平凡因子就不是易验证的。在一般情况下，易验证问题未必是易解的。

(1) 给定一个解易验证问题 P 的蒙特卡罗方法，由此设计一个相应的解问题 P 的拉斯维加斯算法。

(2) 给定一个解易验证问题 P 的拉斯维加斯算法，由此设计一个相应的解问题 P 的蒙特卡罗算法。

分析与解答：

(1) 设给定的解易验证问题 P 的蒙特卡罗算法为 Monte，验证其解的正确性的算法为 charact，则相应的解问题 P 的拉斯维加斯算法 Las 如下：

```
static void Las(ST x) {
    boolean r=Monte(x);
    while(!charact(x,r))r=Monte(x);
```

```
}
```

(2) 设给定的解易验证问题 P 的拉斯维加斯算法为 Las。在超过时间界限时终止算法 Las。相应的解问题 P 的蒙特卡罗算法 Monte 如下：

```
static boolean Monte1(ST x) {
    Las(x);
    if(timeused>maxt)return false;
    else return true;
}
```

习题 7-7　用数组模拟有序链表

用数组模拟有序链表的数据结构，设计支持下列运算的舍伍德型算法，并分析各运算所需的计算时间。

(1) Predecessor：找出一给定元素 x 在有序集 S 中的前驱元素。

(2) Successor：找出一给定元素 x 在有序集 S 中的后继元素。

(3) Min：找出有序集 S 中的最小元素。

(4) Max：找出有序集 S 中的最大元素。

分析与解答：

对主教材中的有序链表类 OrderList 作简单修改。

习题 7-8　$O(n^{3/2})$舍伍德型排序算法

采用数组模拟有序链表的数据结构，设计一个舍伍德型排序算法，使算法最坏情况下的平均计算时间为 $O(n^{3/2})$。

分析与解答：

用教材中的有序链表类 OrderList。对有序链表作 n 次插入运算。第 i 次插入运算平均需要 $O(\sqrt{i})$ 计算时间。因此，相应的排序算法在最坏情况下的平均计算时间为 $O(n^{3/2})$。

习题 7-9　n 后问题解的存在性

如果对于某一个 n 的值，n 后问题无解，则算法将陷入死循环。

(1) 证明或否定下述论断：对于 $n \geqslant 4$，n 后问题有解。

(2) 是否存在一个正数 δ，使得对所有 $n \geqslant 4$ 算法成功的概率至少是 δ。

分析与解答：

用 x_{ij} 表示在棋盘格子(i,j)处放置皇后的状态。当 $x_{ij}=1$ 时，表示在棋盘格子(i,j)处放置了一个皇后，否则没有放置皇后。n 后问题可以表示为如下的 0-1 线性规划问题

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}$$

$$\sum_{j=1}^{n} x_{ij} \leqslant 1 \quad 1 \leqslant i \leqslant n$$

$$\sum_{i=1}^{n} x_{ij} \leqslant 1 \quad 1 \leqslant j \leqslant n$$

$$\sum_{i+j=k} x_{ij} \quad 2 \leqslant k \leqslant 2n$$

$$\sum_{i-j=k} x_{ij} \leqslant 1 \quad 1-n \leqslant k \leqslant n-1$$

$$x_{ij} \in \{0,1\} \quad 1 \leqslant i,j \leqslant n$$

易知，$\max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} \leqslant n$。

(1) 下面对 $n \geqslant 4$ 构造上述线性规划问题的一个解，使 $\max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} = n$，从而证明，对于 $n \geqslant 4$，n 后问题有解。分为以下3种情况：

① 当 $n \geqslant 4$ 为偶数且 $(n-2)\%6>0$ 时，对于 $1 \leqslant j \leqslant n/2$，取 $x(j,2j)=1$；$x(n/2+j, 2j-1)=1$。

② 当 $n \geqslant 4$ 为偶数且 $n\%6>0$ 时，对于 $1 \leqslant j \leqslant n/2$，取 $x(j,k(j))=1$；$x(n+1-j, n-k(j))=1$。其中，$k(j)=(n/2+2(j-1)-1)\%n$。

③ 当 $n>4$ 为奇数时，取 $x(n,n)=1$，转换为 $(n-1)\times(n-1)$ 棋盘的问题，用①或②的方法构造其余的值。

按照上述方法构造 n 后问题解的算法如下：

```
static void construct(int k)
{
    if(k<4)return;
    if (odd(k))x[k]=k--;
    if((k-2)%6>0)build1(k);
    else build2(k);
}

static void build1(int k)
{
    k/=2;
    for(int i=1;i<=k;i++){x[i]=2*i;x[k+i]=2*i-1;}
}

static void build2(int k)
{
    for(int i=1;i<=k/2;i++){
        x[i]=1+(2*(i-1)+k/2-1)%k;
        x[k+1-i]=k-(2*(i-1)+k/2-1)%k;
    }
}
```

容易验证，按照上述方法构造出的解是相应于 n 后问题的0-1线性规划问题的一个解。由此可见，对于 $n \geqslant 4$，n 后问题有解。

(2) 不存在。

习题 7-10　整数因子分解算法

假设已有一个算法 Prime(n) 可用于测试整数 n 是否为一素数。另外还有一个算法 Split(n) 可实现对合数 n 的因子分割。试利用这两个算法设计一个对给定整数 n 进行因子

分解的算法。

分析与解答：

因子分解算法如下：

```
static void fact(int n)
{
    if(prime(n)){output(n);return;}
    int i=split(n);
    if(i>1)fact(i);
    if(n>i)fact(n/i);
}
```

习题 7-11　非蒙特卡罗算法的例子

（1）试证明下面的算法 primality 能以 80%以上的正确率判定给定的一个整数 n 是否为素数。另一方面，举出整数 n 的一个例子表明算法对此整数 n 总是给出错误的解答，进而说明该算法不是一个蒙特卡罗算法。

```
public static boolean primality(int n)
{
    if (gcd(n,30030)==1) return true;
    else return false;
}
```

（2）试找出上述算法 primality 中可用于替换整数 30030 的另一个整数（可使用大整数），使得用此整数代替 30030 后，算法的正确率提高到 85%以上。

分析与解答：

（1）30030 是前 6 个素数的乘积，$30030=2\times3\times5\times7\times11\times13$。因此，当合数含有素因子 2,3,5,7,11,13 时，算法 primality 给出的解答是正确的。而当合数不含素因子 2,3,5,7,11,13 时，算法 primality 给出的解答是错误的。例如，当 $n=323=17\times19$ 时，算法 primality 给出的解答总是 true，总是错误的。可见算法 primality 不是一个蒙特卡罗算法。

一般情况下，全体整数集合中含有素因子 p 的整数的比例为 $1/p$。设前 m 个素数为 $p_1,p_2,\cdots,p_m$，由容斥原理知，全体整数集合中至少含有素因子 $p_1,p_2,\cdots,p_m$ 之一的整数的比例为：

$$\sum_{i=1}^{m}\frac{1}{p_i}-\sum_{i=1}^{m-1}\sum_{j=i+1}^{m}\frac{1}{p_ip_j}+\cdots+(-1)^{m-1}\frac{1}{p_1p_2\cdots p_m}=1-\prod_{i=1}^{m}\left(1-\frac{1}{p_i}\right)$$

对于本题容易计算出，全体整数集合中至少含有前 6 个素因子之一的整数的比例为：

$$1-\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\left(1-\frac{1}{7}\right)\left(1-\frac{1}{11}\right)\left(1-\frac{1}{13}\right)=0.8082$$

由此可见，算法 primality 能以 80.82%以上的正确率判定给定的一个整数 n 是否为素数。

（2）要使算法的正确率提高到 85%以上，必须用前 m 个素数的乘积 $p_1p_2\cdots p_m$ 替代算法中的常数 30030，使 $1-\prod_{i=1}^{m}\left(1-\frac{1}{p_i}\right)>0.85$。

经计算得知：当 $m=11$ 时，$1-\prod_{i=1}^{11}\left(1-\frac{1}{p_i}\right)=0.8471$；当 $m=12$ 时，$1-\prod_{i=1}^{12}\left(1-\frac{1}{p_i}\right)=0.8513$。

可见应该用 $p_1p_2\cdots p_{12}=7420738134810$ 替代算法中的常数 30030 才能使算法的正确率提高到 85%以上。

习题 7-12　重复 3 次的蒙特卡罗算法

设 mc(x)是一个一致的 75%正确的蒙特卡罗算法，考虑下面的算法：

```
public static int mc3(int x)
{
    int t,u,v;
    t=mc(x);
    u=mc(x);
    v=mc(x);
    if ((t==u)||(t==v)) return t;
    return v;
}
```

(1) 试证明上述算法 mc3(x)是一致的$\frac{27}{32}$正确的算法，因此是 84%正确的。

(2) 试证明如果 mc(x)不是一致的，则 mc3(x)的正确率有可能低于 71%。

分析与解答：

(1) 重复 3 次的蒙特卡罗算法各次正确的分布有 8 种不同情况：

000,001,010,011,100,101,110,111

其中，011,101,110,111 这 4 种情况返回正确解。因此，返回正确解的概率为

$$\frac{1}{4}\times\frac{3}{4}\times\frac{3}{4}+\frac{3}{4}\times\frac{1}{4}\times\frac{3}{4}+\frac{3}{4}\times\frac{3}{4}\times\frac{1}{4}+\frac{3}{4}\times\frac{3}{4}\times\frac{3}{4}=\frac{27}{32}$$

(2) 如果 mc(x)不是一致的，则 110 不能保证返回正确解。因此，返回正确解的概率可能低到

$$\frac{1}{4}\times\frac{3}{4}\times\frac{3}{4}+\frac{3}{4}\times\frac{1}{4}\times\frac{3}{4}+\frac{3}{4}\times\frac{3}{4}\times\frac{3}{4}=\frac{45}{64}=0.7031$$

因此，mc3(x)的正确率有可能低于 71%。

习题 7-13　集合随机元素算法

设 $I=\{1,2,\cdots,n\}$，$S\subseteq I$ 是 I 的一个子集。mc(x)是一个偏假 p 正确蒙特卡罗算法。该算法用于判定所给的整数 $1\leqslant x\leqslant n$ 是否为集合 S 中的整数，即 $x\in S$。设 $q=1-p$。由偏假算法的定义可知，对任意 $x\in S$ 有 $\text{Prob}\{\text{mc}(x)=\text{true}\}=1$。当 $x\notin S$ 时，$\text{Prob}\{\text{mc}(x)=\text{true}\}\leqslant q$。

考虑下面的产生 S 中随机元素的算法 genRand：

```
public static boolean repeatMC(int x,int k)
{
    int i=0;
    boolean ans=true;
```

```
        while (ans&&(i<k)) {
            i++;
            ans=mc1(x);
            }
        return ans;
    }
    public static int genRand(int n,int k)
    {
        rnd=new Random();
        int x=rnd.random(n)+1;
        while (!repeatMC(x,k)) x=rnd.random(n)+1;
        return x;
    }
```

假设由语句 $x=$rnd.random(n)+1 产生的整数 $x\in S$ 的概率为 r，证明算法 genRand 返回的整数不在 S 中的概率最多为 $\dfrac{1}{1+\dfrac{r}{1-r}q^{-k}}$。

分析与解答：

算法 genRand 返回的整数 x 是第 1 次由语句 $x=$rnd.random(n)+1 产生的整数，且 x 不在 S 中的概率为 $(1-r)q^k$；算法 genRand 返回的整数 x 是由语句 $x=$rnd.random(n)+1 第 2 次产生的整数，且 x 不在 S 中的概率为 $(1-r)(1-q^k)(1-r)q^k=(1-r)^2(1-q^k)q^k+\cdots$。

由此可知，算法 genRand 返回的整数不在 S 中的概率为

$$
\begin{aligned}
&(1-r)q^k+(1-r)^2(1-q^k)q^k+(1-r)^3(1-q^k)^2q^k+\cdots\\
&=(1-r)q^k\sum_{i=0}^{\infty}(1-r)(1-q^k)^i=\frac{(1-r)q^k}{1-(1-r)(1-q^k)}\\
&=\frac{(1-r)q^k}{1-(1-r)(1-q^k)}=\frac{1}{\dfrac{1}{(1-r)q^k}-\left(\dfrac{1}{q^k}-1\right)}\\
&=\frac{1}{1+\left(\dfrac{1}{1-r}-1\right)q^{-k}}=\frac{1}{1+\dfrac{r}{1-r}q^{-k}}
\end{aligned}
$$

习题 7-14　由蒙特卡罗算法构造拉斯维加斯算法

设算法 A 和 B 是解同一判定问题的两个有效的蒙特卡罗算法。算法 A 是一个 p 正确偏真算法，算法 B 则是一个 q 正确偏假算法。试利用这两个算法设计一个解同一问题的拉斯维加斯算法，并使所得到的算法对任何实例的成功率尽可能高。

分析与解答：

```
    static boolean Las(ST x)
    {
        while(true){
            if(Monte(x))return true;
```

```
        if(!Montel(x))return false;
    }
}
```

习题 7-15　产生素数算法

考虑下面的无限循环算法：

```
public static void printPrimes()
{
    System.out.println('2');
    System.out.println('3');
    int n=5;
    while (true) {
      int m=(int) Math.floor(Math.log((double)n));
      if (primeMC(n,m)) System.out.println(n);
      n=n+2;
      }
}
```

易知，每一个素数都会被上述算法输出。但是除了所有素数外，算法可能偶尔错误地输出某些合数。说明上述情况不太少可能发生。或更精确地，证明上述算法错误地输出1个大于100的合数的概率小于1%。

分析与解答：

$m=\log n$。primeMC(n,m)发生错误的概率小于 $\left(\frac{1}{4}\right)^m=\frac{1}{2^{2\log n}}=\frac{1}{n^2}$。

习题 7-16　矩阵方程问题

给定3个 $n\times n$ 矩阵 $\boldsymbol{a}$，$\boldsymbol{b}$ 和 $\boldsymbol{c}$，下面的偏假 $\frac{1}{2}$ 正确的蒙特卡罗算法用于判定 $\boldsymbol{ab}=\boldsymbol{c}$。

```
public static boolean product(double [][]a, double [][]b, double [][]c, int n)
{// 判定 ab=c 的蒙特卡罗算法
  rnd=new Random();
  double []x=new double [n+1];
  double []y=new double [n+1];
  double []z=new double [n+1];
  for (int i=1;i<=n;i++) {
    x[i]=rnd.random(2);
    if (x[i]==0) x[i]=-1;
    }
  mult(b,x,y,n);
  mult(a,y,z,n);
  mult(c,x,y,n);
  for (int i=1;i<=n;i++)
    if (y[i]!=z[i]) return false;
  return true;
}
```

算法所需的计算时间为 $O(n^2)$。显然当 $\boldsymbol{ab}=\boldsymbol{c}$ 时，算法 product($\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},n$)返回 true。试证明当 $\boldsymbol{ab}\neq\boldsymbol{c}$ 时，算法返回值为 false 的概率至少为 1/2（提示：考虑矩阵 $\boldsymbol{ab}-\boldsymbol{c}$ 并证明当 $\boldsymbol{ab}\neq\boldsymbol{c}$ 时，将该矩阵各行相加或相减最终得到的行向量至少有一半是非零向量）。

分析与解答：

设 $X=\{x\in R^n \mid x_i=\pm1,1\leqslant i\leqslant n\}$；$Y=\{x\in X\mid(\boldsymbol{ab}-\boldsymbol{c})x\neq0\}$；$Z=\{x\in X\mid(\boldsymbol{ab}-\boldsymbol{c})x=0\}$。

当 $\boldsymbol{ab}\neq\boldsymbol{c}$ 时，设 $\boldsymbol{ab}-\boldsymbol{c}$ 的第 i 列非 0，即 $(\boldsymbol{ab}-\boldsymbol{c})e_i\neq0$。

对于任意 $z\in Z$，取 $y\in R^n$ 满足：$y_j=z_j,1\leqslant j\leqslant n,j\neq i$；$y_i=-z_i$，则易知 $y=z-2z_ie_i$。由此可知，

$$(\boldsymbol{ab}-\boldsymbol{c})y=(\boldsymbol{ab}-\boldsymbol{c})(z-2z_ie_i)=-2(\boldsymbol{ab}-\boldsymbol{c})e_i\neq0$$

可见，如此构造出的 $y\in Y$，由不同的 z 构造出的 y 也不同。因此，$|Z|\leqslant|Y|$。

由此可知，当 $\boldsymbol{ab}\neq\boldsymbol{c}$ 时，算法 product($\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},n$)返回值为 false 的概率至少为 1/2。

算法实现题 7-1　模平方根问题

★ 问题描述

设 p 是一个奇素数，$1\leqslant x\leqslant p-1$，如果存在一个整数 y，$1\leqslant y\leqslant p-1$，使得 $x=y^2 \pmod p$，则称 y 是 x 的模 p 平方根。例如，63 是 55 的模 103 平方根。试设计一个求整数 x 的模 p 平方根的拉斯维加斯算法。算法的计算时间应为 $\log p$ 的多项式。

★ 算法设计

设计一个拉斯维加斯算法，对于给定的奇素数 p 和整数 x，计算 x 的模 p 平方根。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 p 和 x。

★ 结果输出

将计算出的 x 的模 p 平方根输出到文件 output.txt。当不存在 x 的模 p 平方根时，输出 0。

输入文件示例	输出文件示例
input.txt	output.txt
103 55	63

分析与解答：

求整数 x 的模 p 平方根的 Tonelli 算法是一个拉斯维加斯算法，描述如下：

```
Tonelli(x,p)
{
    1. 随机选取 g;
    2. 设 p-1=2^s t,t 奇数;
    3. e=0;
    4. for(i=2;i<=s;i++) if ((xg^-e)^((p-1)/2^i) ≠1)e=e+2^i;
    5. h=xg^-e;
    6. b=g^(e/2) h^((t+1)/2);
    7. return b;
}
```

算法实现如下：

```
static long sqrt(long a,long q)
{
    Random rnd=new Random();;
    long i,j,b=1,d=0,g,x=0,y=0,h,k,s=0,t=q-1;
    if ((q%2)==0) return 0;
    while(t%2==0){t/=2;s++;}
    g=rnd.random(q-1)+1;
    exeuclid(g,q,d,x,y);
    if(x<0)x+=q;a%=q;
    long e=0;
    for(i=2,j=2;i<=s;i++,j*=2)
        if(del(a,q,e,x,j)!=1) e+=j;
    for(i=1,h=a;i<=e;i++) {h*=x;h%=q;}
    e/=2;
    for(i=1;i<=e;i++){b*=g;b%=q;}
    for(i=1,k=(t+1)/2;i<=k;i++){b*=h;b%=q;}
    return b;
}
```

其中，exeuclid 是扩展的 euclid 算法，对于整数 a 和 b，计算出 d，x 和 y 使 $d=\gcd(a,b)=ax+by$。算法的计算时间为 $O(\log b)$。del 计算 $(ag^{-e})^{(q-1)/2^j}$。

算法返回正确解的概率为 1/2，所需计算时间为 $O(\log^4 q)$。

可以通过多次调用，提高算法返回正确解的概率。

```
static long sqrtLV(long a,long q)
{
    Random rnd=new Random();
    long k=rnd.random(100)+1;
    for(long i=1;i<=k;i++){
        long r=sqrt(a,q);
        if(r*r%q==a)return r;
    }
    return 0;
}
```

算法实现题 7-2 集合相等问题

★ 问题描述

给定两个集合 S 和 T，试设计一个判定 S 和 T 是否相等的蒙特卡罗算法。

★ 算法设计

设计一个拉斯维加斯算法，对于给定的集合 S 和 T，判定其是否相等。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示集合的大小。接下来的 2 行，每行有 n 个正整数，分别表示集合 S 和 T 中的元素。

★ 结果输出

将计算结论输出到文件 output.txt。集合 S 和 T 相等则输出 Yes，否则输出 No。

输入文件示例	输出文件示例
input.txt	output.txt
3	Yes
2 3 7	
7 2 3	

分析与解答：

类似于主教材中的主元素问题。

算法实现题 7-3 逆矩阵问题

★ 问题描述

给定两个 $n \times n$ 矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，试设计一个判定 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 是否互逆的蒙特卡罗算法。算法的计算时间应为 $O(n^2)$。

★ 算法设计

设计一个蒙特卡罗算法，对于给定的矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，判定其是否互逆。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 为 $n \times n$ 矩阵。接下来的 $2n$ 行，每行有 n 个实数，分别表示矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 中的元素。

★ 结果输出

将计算结论输出到文件 output.txt。矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 互逆则输出 Yes，否则输出 No。

输入文件示例	输出文件示例
input.txt	output.txt
3	Yes
1 2 3	
2 2 3	
3 3 3	
−1 1 0	
1 −2 1	
0 1 −0.666667	

分析与解答：

与算法实现题 7-3 的算法类似。

```
static boolean verse(double [][]A, double [][]B, int n)
{// 判定 AB =I 的蒙特卡罗算法
    Random rnd = new Random();;
    double [ ]x=new double [n+1];
    double [ ]y=new double [n+1];
    double [ ]z=new double [n+1];
    for (int i=1;i<=n;i++) {
```

```
            x[i]=rnd.random(2);
            if (x[i]==0.0) x[i]=-1.0;
        }
        mult(B,x,y,n);mult(A,y,z,n);
        for (int i=1;i<=n;i++)
            if (Math.abs(x[i]-z[i])>0.0001) return false;
        return true;
    }
```

算法实现题 7-4　多项式乘积问题

★ 问题描述

给定阶数分别为 n、m 和 $n+m$ 的多项式 $p(x)$、$q(x)$ 和 $r(x)$。试设计一个判定 $p(x)q(x)=r(x)$ 的偏假 1/2 正确的蒙特卡罗算法，并要求算法的计算时间为 $O(n+m)$。

★ 算法设计

设计一个蒙特卡罗算法，对于给定多项式 $p(x)$、$q(x)$ 和 $r(x)$，判定 $p(x)q(x)=r(x)$ 是否成立。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m、l，分别表示多项式 $p(x)$、$q(x)$ 和 $r(x)$ 的阶数。接下来的 3 行，每行分别有 n、m、l 个实数，分别表示多项式 $p(x)$、$q(x)$ 和 $r(x)$ 的系数。

★ 结果输出

将计算结论输出到文件 output.txt。$p(x)q(x)=r(x)$ 是否成立，则输出 Yes，否则输出 No。

输入文件示例	输出文件示例
input.txt	output.txt
2 1 3	Yes
1 2 3	
2 2	
2 6 10 6	

分析与解答：

多项式的阶为 $k=\max\{mn,l\}$。随机选取 $k+1$ 个实数，测试等式是否成立。

算法实现题 7-5　皇后控制问题

★ 问题描述

在一个 $n\times n$ 个方格组成的棋盘上的任一方格中放置一个皇后，该皇后可以控制所在的行、列以及对角线上的所有方格。

对于给定的自然数 n，在 $n\times n$ 个方格组成的棋盘上最少要放置多少个皇后才能控制棋盘上的所有方格，且放置的皇后互不攻击？

★ 算法设计

设计一个拉斯维加斯算法，对于给定的自然数 n $(1\leqslant n\leqslant 100)$ 计算在 $n\times n$ 个方格组成的棋盘上最少要放置多少个皇后才能控制棋盘上的所有方格，且放置的皇后互不攻击。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

★ **结果输出**

将计算出最少皇后数及最佳放置方案输出到文件 output.txt。文件的第 1 行是最少皇后数,第 2 行是皇后的最佳放置方案。

输入文件示例	输出文件示例
input.txt	output.txt
8	5
	0 3 6 0 0 2 5 8

分析与解答:

与主教材中 n 后问题的拉斯维加斯算法类似。具体算法描述如下:

变量 n 表示皇后个数;数组 x 存储 n 后问题的解。place(k)用于测试将皇后 k 置于第 $x[k]$列的合法性。

```
static boolean place(int k)
{
  if (x[k]>0)
    for (int j=1; j<=k-1; j++)
        if (x[j]>0 && (Math.abs(k-j)==Math.abs(x[j]-x[k]) || x[j]==x[k])) return
        false;
  return true;
}
```

ctrl 用于测试将皇后是否已控制棋盘。

```
static boolean ctrl(int nn)
{
    int i,j,t1,t2,cont=0,xmin=0;
    for (i=1; i<=nn; i++)
        for (j=1; j<=nn; j++) yy[i][j]=0;
    for (i=1; i<=nn; i++) {
        if (x[i]>0) {
            xmin++;
            for (j=1; j<=nn; j++) {yy[i][j]=1;yy[j][x[i]]=1;}
            for(t1=i,t2=x[i];t1>=1 && t2>=1;t1--,t2--)yy[t1][t2]=1;
            for(t1=i,t2=x[i];t1<=nn && t2<=nn;t1++,t2++)yy[t1][t2]=1;
            for(t1=i,t2=x[i];t1>=1 && t2<=nn;t1--,t2++)yy[t1][t2]=1;
            for(t1=i,t2=x[i];t1<=nn && t2>=1;t1++,t2--)yy[t1][t2]=1;
        }
    }
    for (i=1; i<=nn; i++)
        for (j=1; j<=nn; j++)cont += yy[i][j];
    return (cont==nn * nn);
}
```

queensLV(stopVegas)实现在棋盘上随机放置若干个皇后的拉斯维加斯算法。其中,1≤stopVegas≤n 表示随机放置的皇后数。

```
static boolean queensLV(int stopVegas)
{
    int m=stopVegas,count=0,cont=10000;
    while(true){
        int ret=qLV(m,stopVegas);
        if(ret>0) {m=ret-1; count=0;}
        else count++;
        if(count>cont) {while(qLV(m+1,stopVegas)==0);break;}
    }
    return true;
}

static int qLV(int m,int stopVegas)
{
    Random rnd=new Randomf();
    while(true){
        int k=1;
        while (k<=stopVegas) {
            int count=0;
            for (int i=0; i<=n; i++) {
                x[k]=i;
                if (place(k)) y[count++]=i;
            }
            x[k++]=y[rnd.random(count)];
        }
        int pla=placed(stopVegas);
        if(pla<=m) return pla;
        else return 0;
    }
}
```

与回溯法相结合的解 n 后控制问题的拉斯维加斯算法描述如下:

```
static boolean nQueen(int nn)
{
    n=nn;
    int []p=new int [n+1];
    int []q=new int [n+1];
    int [][]r=new int[n+1][n+1];
    for (int i=0; i<=n; i++) p[i]=0;
    x=p;y=q;yy=r;
    int stop=3;
    if(n>15)stop=n-15;
    boolean found=false;
```

```
    while (!queensLV(stop));
    if(backtrack(stop+1)){
        System.out.println(placed(n));
        for (int i=1;i<=n;i++) System.out.print(p[i]+" ");
        System.out.println();
        found=true;
    }
    return found;
}
```

算法的回溯搜索部分与解 n 后问题的回溯法是类似的，只要找到一个解就返回。

```
static boolean backtrack (int t)
{
    if (t>n ) {
        if (ctrl(n)) return true;
        else return false;}
    for (int i=0; i<=n; i++) {
        x[t]=i;
        if (place(t) && backtrack(t+1))return true;
    }
    return false;
}
```

placed 计算已放置的皇后数。

```
static int placed(int k)
{
    int num=0;
    for (int j=1; j<=k; j++)if (x[j]>0) num++;
    return num;
}
```

算法实现题 7-6　3-SAT 问题

★ 问题描述

SAT 的一个实例是 k 个布尔变量 $x_1,x_2,\cdots,x_k$ 的 m 个布尔表达式 $A_1,A_2,\cdots,A_m$。若存在各布尔变量 x_i(其中 $1\leqslant i\leqslant k$)的 0,1 赋值，使每个布尔表达式 A_i(其中 $1\leqslant i\leqslant m$)都取值 1，则称布尔表达式 $A_1A_2\cdots A_m$ 是可满足的。

◆ **合取范式的可满足性问题 CNF-SAT**

如果一个布尔表达式是一些因子和之积，则称之为合取范式，简称 CNF(conjunctive normal form)。这里的因子是变量 x 或 x。例如，$(x_1+x_2)(x_2+x_3)(x_1+x_2+x_3)$ 就是一个合取范式，而 $x_1x_2+x_3$ 就不是合取范式。

◆ ***k*-SAT**

如果一个布尔合取范式的每个乘积项最多是 k 个因子的析取式，就称之为 k 元合取范式，简记为 k-CNF。一个 k-SAT 问题是判定一个 k-CNF 是否可满足。特别地，当 $k=3$ 时，3-SAT 问题在 NP 完全问题树中具有重要地位。

◆ **MAX-SAT**

给定 k 个布尔变量 $x_1, x_2, \cdots, x_k$ 的 m 个布尔表达式 $A_1, A_2, \cdots, A_m$，求各布尔变量 x_i（其中 $1 \leqslant i \leqslant k$）的 0,1 赋值，使尽可能多的布尔表达式 A_i 取值 1。

◆ **Weighted-MAX-SAT**

给定 k 个布尔变量 $x_1, x_2, \cdots, x_k$ 的 m 个布尔表达式 $A_1, A_2, \cdots, A_m$，每个布尔表达式 A_i 都有一个权值 w_i，求各布尔变量 x_i（其中 $1 \leqslant i \leqslant k$）的 0,1 赋值，使取值 1 的布尔表达式权值之和达到最大。

★ **算法设计**

对于给定的带权 3-CNF，设计一个蒙特卡罗算法，使其权值之和尽可能大。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m，分别表示变量数和布尔表达式数。接下来的 m 行中，每行有 5 个整数 $w, i, j, k, 0$，表示相应表达式的权值为 w，表达式含的变量下标分别为 i, j, k，行末以 0 结尾。下标为负数时，表示相应的变量为取反变量。

★ **结果输出**

将计算出的最大权值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 3	26
9 3 1 4 0	
9 1 −5 3 0	
8 2 −5 1 0	

分析与解答：

与主教材中 n 后问题的拉斯维加斯算法类似。随机产生布尔变量的真值赋值。

算法实现题 7-7　战车问题

★ **问题描述**

在 $n \times n$ 格的棋盘上放置彼此不受攻击的车。按照国际象棋的规则，车可以攻击与之处在同一行或同一列上的棋子。在棋盘上的若干个格中设置了堡垒，战车无法穿越堡垒攻击别的战车。对于给定的设置了堡垒的 $n \times n$ 格棋盘，设法放置尽可能多彼此不受攻击的车。

★ **算法设计**

对于给定的设置了堡垒的 $n \times n$ 格棋盘，设计一个概率算法，在棋盘上放置尽可能多彼此不受攻击的车。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。接下来的 n 行中，每行有 1 个由字符.和 X 组成的长度为 n 的字符串。

★ **结果输出**

将计算出的在棋盘上可以放置的彼此不受攻击的战车数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4	6
....	
..X.	
.X..	
....	

分析与解答：

与主教材中 n 后问题的拉斯维加斯算法类似。在 $n\times n$ 格的棋盘上随机放置彼此不受攻击的车。具体算法实现如下。

init 作初始化计算。

```
static void init(int n)
{
    int i,j,k,x,y;
    state=new int[n*n+2];
    link=new int[n*n+1][2*n+1];
    state[0]=-1;
    state[n*n+1]=-1;
    for (int no=1; no<=n*n; no++) {
        i=(no-1)/n;j=(no-1)%n;
        link[no][0]=0;state[no]=-1;
        if (row[i].charAt(j)== '.') {
            state[no]=0;k=0;y=j;
            while ((y<n-1) && (row[i].charAt(y+1) == '.')) {
                link[no][0]++;y++;k++;
                link[no][k]=i*n+y+1;
            }
            y=j;
            while ((y>0) && (row[i].charAt(y-1) == '.')) {
                link[no][0]++;y--;k++;
                link[no][k]=i*n+y+1;
            }
            x=i;
            while ((x<n-1) && (row[x+1].charAt(j) == '.')) {
                link[no][0]++;x++;k++;
                link[no][k]=x*n+j+1;
            }
            x=i;
            while ((x>0) && (row[x-1].charAt(j)== '.')) {
                link[no][0]++;x--;k++;
                link[no][k]=x*n+j+1;
            }
        }
    }
```

```
}
```

实现随机算法的主函数如下：

```
public static void main(String [ ] args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    Random rnd=new Random();;
    for (int i=0; i<n; i++) row[i]=keyboard.readString();
    init(n);
    int max=0,rept=0,put=0;
    while(rept<100000){
        rept++;int count=0;
        while(true){
            int x=rnd.random(n*n)+1, c=x;
            while ((x<=n*n) && (state[x]!=put))x++;
            if (state[x]!=put){
                x=c;
                while ((x>0) && (state[x]!=put))x--;
            }
            if (state[x]==put){
                count++;
                for (int i=1; i<=link[x][0]; i++)
                    if (state[link[x][i]]==put) state[link[x][i]]++;
                state[x]++;
            }
            else break;
        }
        if (count>max) max=count;
        put++;
    }
    System.out.println(max);
}
```

算法实现题 7-8　圆排列问题

★ 问题描述

给定 n 个大小不等的圆 $c_1,c_2,\cdots,c_n$，现要将这n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 $n=3$，且所给的3个圆的半径分别为 1,1,2 时，这 3 个圆的最小长度的圆排列如图 7-1 所示。其最小长度为 $2+4\sqrt{2}$。

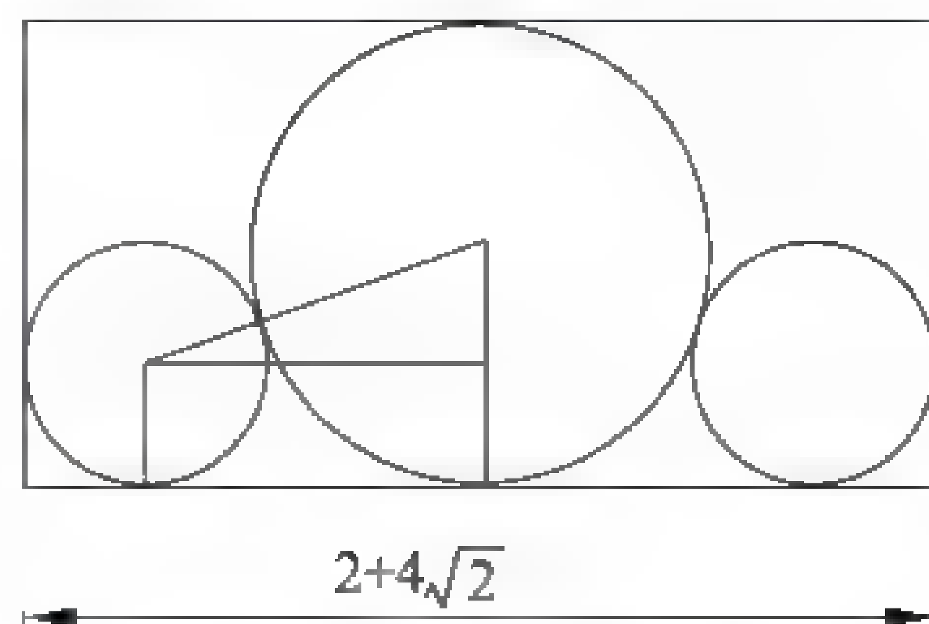


图 7-1　圆排列问题

解圆排列问题的一个随机化算法如下：

```
static void Circle_search(int [ ]x,int n)
{
```

```
    random_perm(x);
    boolean found=true;
    while(found){
        found=false;
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                if(MyMath.swap(x,i,j) reduces length){
                    MyMath.swap(x,i,j);
                    found=true;
                }
    }
}
```

其中，random_perm(x)产生 x 的一个随机排列。

★ 算法设计

根据上述算法框架，设计一个随机化算法，对于给定的 n 个圆，计算 n 个圆的最佳排列方案，使其长度尽可能小。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，$1\leqslant n\leqslant 20$。第 2 行有 n 个数，表示 n 个圆的半径。

★ 结果输出

将计算出的最小圆排列的长度输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	7.65685
1 1 2	

分析与解答：

与主教材中 n 后问题的拉斯维加斯算法类似，算法已在题目描述中。

算法实现题 7-9 骑士控制问题

★ 问题描述

在一个 $m\times n$ 个方格的国际象棋棋盘上，马(骑士)可以攻击的棋盘方格如图 7-2 所示。

★ 算法设计

对于给定的 $m\times n$ 个方格的国际象棋棋盘，计算棋盘上最少需要放置多少个骑士，使得每个方格至少受到 k 个骑士的攻击。

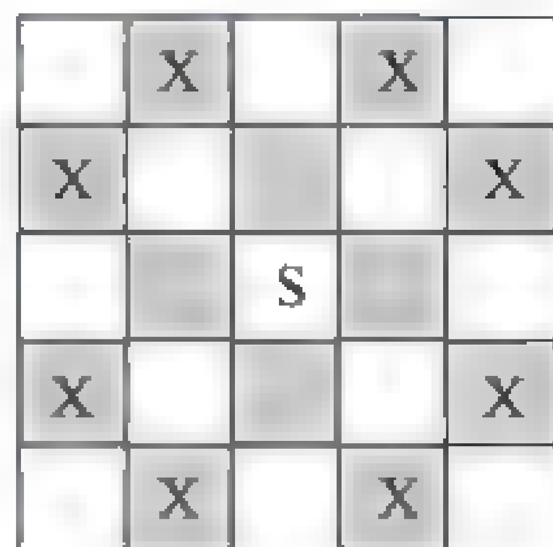


图 7-2 马攻击的棋盘方格

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 m，n 和 k，分别表示棋盘的大小为 m 行，n 列；每个方格至少受到 k 个骑士的攻击。

★ 结果输出

将计算出的最少骑士数输出到文件 output.txt。问题无解时输出“No solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4 4 1	6

分析与解答：

用主教材中 n 后问题的拉斯维加斯算法类似的算法求解。

此题也可以用整数线性规划算法求解。设第 i 行第 j 列相应的变量为 x_{ij}。变量 x_{ij} 的含义是，(i,j)方格中放置了 x_{ij} 个骑士，$x_{ij} \in \{0,1\}$。

骑士控制问题的求解目标是 $\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}$ 达到最小。

约束条件是，每个方格至少受到 k 个骑士的攻击。

设可以攻击(i,j)方格的其他方格的集合是 S_{ij}，$k_{ij} = |S_{ij}|$，则 $2 \leqslant k_{ij} \leqslant 8$。

相应于每个方格的约束条件可以表述为 $\sum_{(p,q)\in S_{ij}} x_{pq} \geqslant k$。

由此可见，骑士控制问题可以变换为如下整数线性规划问题

$$\min \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}$$

$$\text{s.t.} \quad \sum_{(p,q)\in S_{ij}} x_{pq} \geqslant k$$

$$x_{ij} \in \{0,1\}$$

算法实现题 7-10　骑士对攻问题

★ 问题描述

在一个 $m\times n$ 个方格的国际象棋棋盘上，马(骑士)可以攻击的棋盘方格如图 7-3 所示。

★ 算法设计

对于给定的 $m\times n$ 个方格的国际象棋棋盘，计算棋盘上最多可以放置多少个骑士，使得每个骑士仅受到另一个骑士的攻击，如图 7-4 所示。

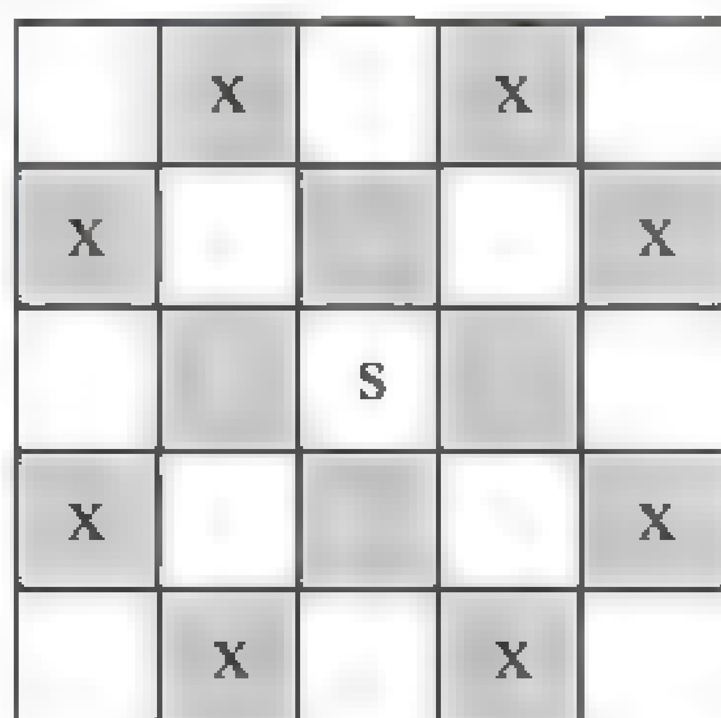


图 7-3　国际象棋棋盘

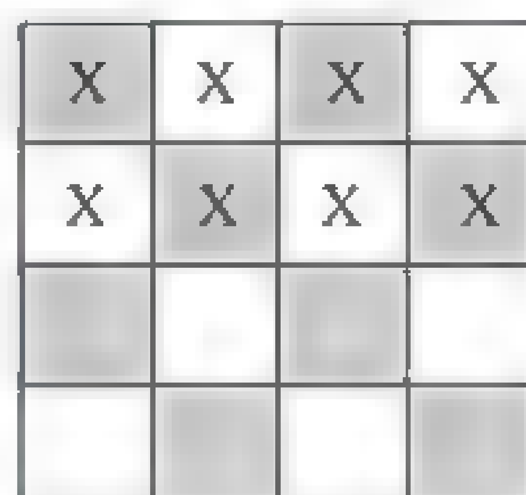


图 7-4　骑士对攻问题

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n，分别表示棋盘的大小为 m 行，n 列。

★ 结果输出

将计算出的最多骑士数输出到文件 output.txt。问题无解时，则输出“No solution!”。

输入文件示例	输出文件示例
input. txt	output. txt
4 4	8

分析与解答：

用主教材中 n 后问题的拉斯维加斯算法类似的算法求解。

此题也可以用整数线性规划算法求解。设第 i 行第 j 列相应的变量为 x_{ij}。变量 x_{ij} 的含义是，(i,j) 方格中放置了 x_{ij} 个骑士，$x_{ij} \in \{0,1\}$。

骑士对攻问题的求解目标是 $\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}$ 达到最大。

约束条件是，每个骑士仅受到另一个骑士的攻击。

设可以攻击 (i,j) 方格的其他方格的集合是 S_{ij}，$k_{ij} = |S_{ij}|$，则 $2 \leqslant k_{ij} \leqslant 8$。

相应于每个方格的约束条件可以表述为

$$(k_{ij} - 1)x_{ij} + \sum_{(p,q) \in S_{ij}} x_{pq} \leqslant k_{ij}$$

$$\sum_{(p,q) \in S_{ij}} x_{pq} - x_{ij} \geqslant 0$$

由此可见，骑士对攻问题可以变换为如下整数线性规划问题

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \\
\text{s.t.} \quad & (k_{ij} - 1)x_{ij} + \sum_{(p,q) \in S_{ij}} x_{pq} \leqslant k_{ij} \\
& \sum_{(p,q) \in S_{ij}} x_{pq} - x_{ij} \geqslant 0 \\
& x_{ij} \in \{0,1\}
\end{aligned}$$

第 8 章 NP 完全性理论与近似算法

习题 8-1　析取范式的可满足性

证明析取范式的可满足性问题属于 P 类。

分析与解答：

析取范式 α 是由因子之和的和式给出的。这里因子是指变量 x 或 $\bar{x}$，每个因子之积称为一个析取式。例如，$x_1x_2+x_2x_3+\bar{x}_1\bar{x}_2x_3$ 就是一个析取范式。

设给定一个析取范式 $\alpha=\sum_{i=1}^{m} f_i(x_1,x_2,\cdots,x_n)$，其中，$f_i(x_1,x_2,\cdots,x_n)$ 是变量 x_j 或 x_j，(其中 $j=1\sim n$) 的一个析取式，$i=1\sim m$。α 是可满足的，当且仅当存在 i，$1\leqslant i\leqslant m$，使 $f_i(x_1,x_2,\cdots,x_n)$ 是可满足的。如果有一个多项式时间算法能判断任何一个析取式的可满足性，则用该算法对 α 的每个析取项逐一判断就可以在多项式时间内确定析取范式 α 的可满足性。因此，问题简化为找一个确定析取式可满足性的多项式时间算法。

设析取式 $f(x_1,x_2,\cdots,x_n)=\alpha_1\alpha_2\cdots\alpha_k$，其中 $\alpha_i\in\{x_j,\bar{x}_j \mid 1\leqslant j\leqslant n\}$。

可以设计在 $O(n+k)$ 时间内确定析取式 $f(x_1,x_2,\cdots,x_n)=\alpha_1\alpha_2\cdots\alpha_k$ 可满足性的算法。因此，析取范式的可满足性问题是多项式时间可解的，即它属于 P 类。

习题 8-2　2-SAT 问题的线性时间算法

2-SAT 是每个合取项恰有两个因子的可满足性问题。证明 2-SAT$\in$P，并提出解此问题的尽可能高效的算法。

分析与解答：

设 2 元合取范式为 $\theta=\prod_{i=1}^{m}(\alpha_i+\beta_i)$，其中，$\alpha_i,\beta_i\in\{x_1,\bar{x}_1,\cdots,x_n,\bar{x}_n\}$，$i=1,2,\cdots,m$。

对于所给的 2 元合取范式 θ，构造相应的有向图 $G=(V,E)$ 如下：

变量 x_i 对应于图 G 的顶点 v_{2i-1}；变量 x_i 对应于图 G 的顶点 v_{2i}，$i=1,2,\cdots,n$，因此，$|V|=2n$。

每一个合取项 $\alpha_i+\beta_i$ 对应于 E 中两条有向边 (u,v) 和 (s,t) 如下：

$$(u,v)=\begin{cases}(v_{2j},v_{2k-1}) & (\alpha_i,\beta_i)=(x_j,x_k)\\(v_{2j},v_{2k}) & (\alpha_i,\beta_i)=(x_j,x_k)\\(v_{2j-1},v_{2k-1}) & (\alpha_i,\beta_i)=(x_j,x_k)\\(v_{2j-1},v_{2k}) & (\alpha_i,\beta_i)=(x_j,x_k)\end{cases}$$

$$(s,t) = \begin{cases} (v_{2k}, v_{2j-1}) & (\alpha_i, \beta_i) = (x_j, x_k) \\ (v_{2k-1}, v_{2j-1}) & (\alpha_i, \beta_i) = (x_j, x_k) \\ (v_{2k}, v_{2j}) & (\alpha_i, \beta_i) = (x_j, x_k) \\ (v_{2k-1}, v_{2j}) & (\alpha_i, \beta_i) = (x_j, x_k) \end{cases}$$

显而易见，$|E| = 2m$。边$(v_i, v_j) \in E$ 表示顶点 v_i 相应的变量取值 1，则顶点 v_j 相应的变量也应取值 1，否则 $\theta = 0$。

对于给定的 2 元合取范式 θ，可以在 $O(m+n)$ 时间内构造出上述有向图 G。可以证明，θ 是可满足的当且仅当图 G 中不存在形如 $x_i \to \cdots \to \bar{x}_i \to \cdots \to x_i$ 这样的圈。而图 G 中的这种圈可以通过求 G 的强连通分支来检测。用求图的强连通分支的算法，可在 $O(m+n)$ 时间内找出图 G 的所有强连通分支。对于每对变量 x_i 和 $\bar{x}_i$ 判定是否属于 G 的同一个强连通分支。若属于 G 的同一个强连通分支，则说明 G 中存在 $x_i \to \cdots \to \bar{x}_i \to \cdots \to x_i$ 的圈，因此 θ 不可满足。若 G 的每一个强连通分支都不存在这样的圈，则 θ 是可满足的。上述判定显然只需 $O(n)$ 时间。

综上所述，2 元合取范式 θ 的可满足性可以在 $O(m+n)$ 时间内判定。

习题 8-3　整数规划问题

给定一个 $m \times n$ 整数矩阵 $\boldsymbol{A}$ 和一个 m 元整数向量 $\boldsymbol{b}$，判定是否存在一个 n 元 0-1 向量 $\boldsymbol{x}$，使得 $\boldsymbol{Ax} \leqslant \boldsymbol{b}$。该问题称为 0-1 整数规划问题。证明该问题是 NP 完全问题。（提示：证明 3-SAT$\propto_p$0-1 整数规划问题）

分析与解答：

记 0-1 整数线性规划问题为 0-1-ILP。对于给定的 n 元 0-1 向量 $\boldsymbol{x}$，显然可在多项式时间内验证 $\boldsymbol{Ax} \leqslant \boldsymbol{b}$。因此，0-1-ILP$\in$NP。

下面通过证明 3-SAT $\propto_p$0-1-ILP 来证明 0-1-ILP 的 NP 完全性。

设 $\Phi = C_1 C_2 \cdots C_k$ 是一个 3 元合取范式，其中，

$$C_r = l_1^r + l_2^r + l_3^r = (a_1^r, a_2^r, \cdots, a_{2n}^r)(x_1, x_2, \cdots x_n, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)^{\mathrm{T}}$$

当 $l_j^r = x_i$ 时，$a_i^r = 1$；当 $l_j^r = \bar{x}_i$ 时，$a_{n+i}^r = 1, j = 1,2,3$；其余的诸 a_i^r 均为 0。

Φ 可满足当且仅当存在 x_i 的真值赋值，使 $C_r = 1, r = 1 \sim k$。若将 x_i 看作 0-1 变量，则应有

$$(\boldsymbol{a}^r)^{\mathrm{T}} \begin{pmatrix} \boldsymbol{X} \\ \boldsymbol{I} - \boldsymbol{X} \end{pmatrix} \geqslant 1, \quad r = 1 \sim k$$

其中，$\boldsymbol{a}^r = (a_1^r, a_2^r, \cdots, a_{2n}^r)^{\mathrm{T}}, \boldsymbol{X} = (x_1, \bar{x}_2, \cdots, x_n)^{\mathrm{T}}, \boldsymbol{I} = (1,1,\cdots,1)^{\mathrm{T}}$。

令 $\boldsymbol{B} = \begin{bmatrix} (a^1)^{\mathrm{T}} \\ (a^2)^{\mathrm{T}} \\ \vdots \\ (a^k)^{\mathrm{T}} \end{bmatrix}$，则有 $\boldsymbol{B} \begin{pmatrix} \boldsymbol{X} \\ \boldsymbol{I} - \boldsymbol{X} \end{pmatrix} \geqslant \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$。

其中，$\boldsymbol{B}$ 是一个 $k \times (2n)$ 矩阵。将 $\boldsymbol{B}$ 分块为 2 个 $k \times n$ 矩阵 $\boldsymbol{B}_1$ 和 $\boldsymbol{B}_2$，即 $\boldsymbol{B} = (\boldsymbol{B}_1, \boldsymbol{B}_2)$，则有

$$\boldsymbol{B}_1 \boldsymbol{X} + \boldsymbol{B}_2 (\boldsymbol{I} - \boldsymbol{X}) \geqslant \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

$$(\boldsymbol{B}_2 - \boldsymbol{B}_1)\boldsymbol{X} \leqslant \boldsymbol{B}_2\boldsymbol{I} - \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

取 $\boldsymbol{A} = \boldsymbol{B}_2 - \boldsymbol{B}_1$，$\boldsymbol{b} = \boldsymbol{B}_2\boldsymbol{I} - \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$，$\boldsymbol{AX} \leqslant \boldsymbol{b}$。

易见，$\boldsymbol{A}$ 为整数矩阵，$\boldsymbol{b}$ 为整数向量，Φ 可满足当且仅当 $\boldsymbol{AX} \leqslant \boldsymbol{b}, x_i \in \{0,1\}, 1 \leqslant i \leqslant n$ 有解。

以上的变换显然可在多项式时间内完成，因此，3-SAT$\propto_p$0-1-ILP。

由 3-SAT$\in$NPC 即知，0-1-ILP$\in$NPC。

习题 8-4　划分问题

给定整数集合 S，判定 S 是否可划分为两个子集 A 和 $\overline{A}$（$=S-A$），使得 $\sum\limits_{x\in A} x = \sum\limits_{x\in \overline{A}} x$。证明该问题是 NP 完全问题。

分析与解答：

记划分问题为 PARTITION。对于 S 的给定的划分 A 和 $\overline{A}$，显然可在多项式时间内验证 $\sum\limits_{x\in A} x = \sum\limits_{x\in \overline{A}} x$。因此，PARTITION$\in$NP。

下面证明 SUBSET-SUM$\propto_p$PARTITION。

设子集和问题的一个输入为 $X=\{x_1,x_2,\cdots,x_n\}$，$i=1\sim n$ 为正整数，和一个正整数 t。子集和问题要求判定是否存在 $I\subseteq\{1,2,\cdots,n\}$，使得 $\sum\limits_{i\in I} x_i = t$。

对于子集和问题的输入，取 $m_1 = 2\sum\limits_{i=1}^{n} x_i$，$m_2 = \sum\limits_{i=1}^{n} x_i + 2t$，$S=\{x_1,x_2,\cdots,x_n,m_1,m_2\}$ 作为划分问题 PARTITION 的输入。可以证明 X 和 t 是 SUBSET-SUM 的一个 Yes 实例当且仅当相应的 S 是 PARTITION 的一个 Yes 实例。

事实上，当 X 和 t 是 SUBSET-SUM 的一个 Yes 实例时，存在 $I\subseteq\{1,2,\cdots,n\}$，使得 $\sum\limits_{i\in I} x_i = t$，将 S 划分为 A 和 $\overline{A}$ 如下：

$$A=\{x_i \mid i\in I\}\cup\{m_1\},\quad A=\{x_i \mid i\notin I\}\cup\{m_2\}$$

则

$$\sum_{x\in A} x = \sum_{i\in I} x_i + 2\sum_{i=1}^{n} x_i = t + 2\sum_{i=1}^{n} x_i$$

而

$$\sum_{x\in \overline{A}} x = \sum_{i\notin I} x_i + \sum_{i=1}^{n} x_i + 2t = \sum_{i\notin I} x_i + \sum_{i\in I} x_i + \sum_{i=1}^{n} x_i + t = t + 2\sum_{i=1}^{n} x_i$$

因此，按此划分有 $\sum\limits_{x\in A} x = \sum\limits_{x\in A} x$，即 S 是 PARTITION 的一个 Yes 实例。

反之，设 S 是 PARTITION 的一个 Yes 实例，则存在 S 的一个划分 A 和 A 使 $\sum\limits_{x\in A} x = \sum\limits_{x\in A} x$。显然，$m_1$ 和 m_2 应分别属于 A 和 A 之一。不妨设 $m_1\in A$，$m_2\in A$。

取 $I=\{i \mid x_i\in A\}$，则

$$\sum_{i \in I} x_i + m_1 = \sum_{i \notin I} x_i + m_2$$

$$\sum_{i \in I} x_i + 2\sum_{i=1}^{n} x_i = \sum_{i \notin I} x_i + \sum_{i=1}^{n} x_i + 2t$$

$$\sum_{i \in I} x_i + \sum_{i=1}^{n} x_i - \sum_{i \notin I} x_i = 2t$$

$$2\sum_{i \in I} x_i = 2t$$

因此，$\sum_{i \in I} x_i = t$，即 X 和 t 是 SUBSET SUM 的一个 Yes 实例。以上变换显然可以在 $O(|X|)$时间内完成，故 SUBSET SUM$\propto_p$PARTITION。由 SUBSET-SUM∈NPC 即知，PARTITION∈NPC。

习题 8-5　最长简单回路问题

最长简单回路问题是确定给定图 $G=(V,E)$中一条长度最大的简单回路(其中没有重复出现的顶点)的问题。证明最长简单回路问题是 NP 完全问题。

分析与解答：

事实上，可以证明更强的结论，最长简单回路问题是 NP 完全问题。

将哈密顿回路问题 HAM-CYCLE 多项式时间变换为最长简单路问题如下：

```
Hamiltonian(G)
{
    对图 G 的每条边(u,v),

    If(Longest(u,v)==n-1) return true;
    Else return false;
}
```

上述算法调用 m 次最长简单路算法 Longest。由此可知，HAM-CYCLE$\propto_p$LONGEST。容易验证 LONGEST∈NP。由于哈密顿回路问题是 NP 完全问题，故最长简单路问题也是 NP 完全问题。

习题 8-6　平面图着色问题的绝对近似算法

设问题 P 关于实例 I 的精确解为 $c^*(I)$，解问题 P 的近似算法 A 对于实例 I 得到的近似解为 $c(I)$。如果存在一常数 k，使得对于 P 的任何实例 I 均有 $|c^*(I)-c(I)|\leq k$，则称算法 A 是解问题 P 的绝对近似格式。

平面图的色数问题是对于给定的平面图 $G=(V,E)$，确定对其顶点着色的最小色数。试设计解平面图着色问题的一个多项式时间绝对近似算法 A 使得 $|c^*(I)-c(I)|\leq 1$。

分析与解答：

对于给定的平面图 $G=(V,E)$，求 G 的色数的绝对近似算法描述如下：

```
int acolor()
{
    if(G 的顶点集 V 为空) return 0;
    if(G 的边集 E 为空) return 1;
    if(G 是二分图) return 2;
```

```
    return 4;
```

由平面图的4色定理,上述算法只在图 G 是可3着色时,近似地返回4,其余情况下均返回正确解答。因此,对于上述近似算法有 $|c^*(I)-c(I)|\leqslant 1$。

习题 8-7 最优程序存储问题

设有 n 个程序 $1,2,\cdots,n$,要存入两张容量为 maxM 的磁盘中。第 i 个程序需要的存储空间为 m_i,$1\leqslant i\leqslant n$。设计一个算法计算出这两张磁盘能存放的最多程序个数。

(1) 证明上述问题是 NP 难的。

(2) 下面的算法 pStore 是解上述问题的一个绝对近似算法。

```
int pStore(int n, int maxM, int [] m)
{
        sort(m,n);  // 将 m 从小到大排序
        int i=1;
        for (int j=1;j<=2;j++) {
          int sum=0;
        while (sum+m[i]<= maxM) {
          System.out.println("Store program"+i+"on disk  "+j);
          sum+=m[i];
          if (i==n) return i;
          else i++;
          }
    }
  return i-1;
}
```

试证明对于上述算法 pStore,有 $|c^*(I)-c(I)|\leqslant 1$。

分析与解答:

(1) 将划分问题变换为最优程序存储问题。对于划分问题的任一实例 $\{a_1,a_2,\cdots,a_n\}$,不妨设 $\sum_{i=1}^{n} a_i = 2T$。定义相应的最优程序存储问题的实例如下:$\text{Max}=T$;$m_i=a_i$,$1\leqslant i\leqslant n$。显而易见,$\{a_1,a_2,\cdots,a_n\}$有一个划分,当且仅当所有 n 个程序能存入两张盘中。

上述变换只要线性时间。由划分问题的 NP 完全性可知,最优程序存储问题是 NP 难的。

(2) 假设算法 pStore 的解答为 c,而最优值为 c^*。不失一般性,设 $m_1\leqslant m_2\leqslant\cdots\leqslant m_n$。由习题 4-8 的结论可知,如果只有一张容量为 2Max 的磁盘,按照贪心策略可以得到最优解。假设用贪心策略最多可存放 p 个程序到一张容量为 2Max 的磁盘上。显然,$c^*\leqslant p$,且 $\sum_{i=1}^{p} m_i \leqslant 2\text{Max}$。

设 $j=\max\left\{k \mid \sum_{i=1}^{k} m_i \leqslant \text{Max}\right\}$,则 $j\leqslant p$,$\sum_{i=1}^{j+1} m_i > \text{Max}$。算法 pStore 将前 j 个程序存放到第1张磁盘上。由 $\sum_{i=j+1}^{p-1} m_i \leqslant \sum_{i=j+2}^{p} m_i \leqslant \text{Max}$ 可知,算法 pStore 至少将程序 $j+1,\cdots,$

$p-1$ 存放到第 2 张磁盘上。由此可见，$c \geqslant p-1$。因此，$|c^*-c| \leqslant 1$。

习题 8-8　树的最优顶点覆盖

设计一个有效的贪心算法，使其能在线性时间内找到一棵树的最优顶点覆盖。

分析与解答：

设给定的树 T 有 n 个顶点。首先对树 T 作前序标号如下：

(1) 任选一个顶点 r 作为树 T 的根结点。

(2) 对以 r 为根的树作前序遍历，并且在遍历过程中对访问的顶点依次编号。

(3) 用数组 parent 记录每个结点的父结点编号。

树 T 的这个表示过程显然可在 $O(n)$ 时间内完成。前序标号表示法实际上是树在前序标号意义下的父亲数组表示法，具有以下性质：

- 对于 $i=2,\cdots,n$，有 $i>\text{parent}[i]$，当 $i=1$ 时，$\text{parent}[i]=1$；
- 若将树 T 看作是一般的图 $G=(V,E)$，则有 $V=\{1,2,\cdots,n\}$，$E=\{(i,\text{parent}[i]),i=2,\cdots,n\}$；
- 对于任意 j，$2\leqslant j\leqslant n$，定义树 T 的子树 $T_j=(V_j,E_j)$ 为：$V_j=\{1,2,\cdots,j\}$，$E_j=\{(i,\text{parent}[i]),i=2,\cdots,j\}$，则 $\text{parent}[1..j]$ 是子树 T_j 的前序标号表示。特别地，$T_n=T$；
- 标号为 j 的结点是 T_j 的叶结点，$j=2,\cdots,n$。特别地，标号为 n 的结点是 T 的叶结点。

基于树的前序标号表示法，可设计树 T 的最小顶点覆盖的贪心算法 treecover 如下：

```
void treecover()
{
    for(int i=1;i<=n;i++){cover[i]=0;s[i]=0;}
    for(i=n;i>1;i--)
        if((!cover[i]) && (!cover[parent[i]]))s[parent[i]]=1;
}
```

算法 treecover 是一个贪心算法。算法中用数组 cover 来标记选入覆盖点集的树结点，即当结点 i 被选入覆盖点集，则 $\text{cover}[i]=1$，否则 $\text{cover}[i]=0$。

为了说明算法的正确性，必须证明关于树的最小顶点覆盖问题满足贪心选择性质并具有最优子结构性质。

1) 贪心选择性质

对于树 T，存在一个 T 的最小顶点覆盖 S，使 $\text{parent}[n]\in S$。

事实上，对于 T 的任意一个最小顶点覆盖 S，若 $\text{parent}[n]\notin S$，则 $n\in S$，否则 S 就不是 T 的一个顶点覆盖。在这种情况下，令 $S'=S\cup\{\text{parent}[n]\}-\{n\}$，则 S' 仍为 T 的一个顶点覆盖，且 $|S'|=|S|$。因此，S' 是 T 的一个最小顶点覆盖，且 $\text{parent}[n]\in S'$。

2) 最优子结构性质

对于 T 的任一最小顶点覆盖 S，当 $n\in S$ 时，$S-\{n\}$ 是 T_{n-1} 的最小顶点覆盖。

事实上，$S-\{n\}$ 显然是 T_{n-1} 的一个顶点覆盖。若它不是 T_{n-1} 的最小顶点覆盖，则存在 T_{n-1} 的一个更小的顶点覆盖 S'，且 $|S'|<|S|-1$。$S'\cup\{n\}$ 显然是 T 的一个顶点覆盖，且 $|S'\cup\{n\}|\leqslant|S'|+1<|S|$，这与 S 是 T 的一个最小顶点覆盖矛盾。

当 $n \notin S$ 时，设 $i = \text{parent}[n]$，则必有 $i \in S$。

令 $S_1 = \{j \mid i \leqslant j \leqslant n\} \cap S$，则 $S - S_1$ 是 T_{i-1} 的一个最小顶点覆盖。事实上，由于 S 是 T 的一个顶点覆盖，故 $S - S_1$ 是 T_{i-1} 的一个顶点覆盖。若在 T_{i-1} 中有一个比 $S - S_1$ 更小的顶点覆盖 S_{i-1}，则 $|S_{i-1}| < |S - S_1|$。显而易见，$S_{i-1} \cup S_1$ 是 T 的一个顶点覆盖，且 $|S_{i-1} \cup S_1| \leqslant |S_{i-1}| + |S_1| < |S|$，这与 S 是 T 的一个最小顶点覆盖相矛盾。

根据上述的贪心选择性质和最优子结构性质，容易用数学归纳法证明算法 treecover 的正确性。

算法的 for 循环显然只需要 $O(n)$ 时间，从而整个算法所需的时间为 $O(n)$。

习题 8-9　顶点覆盖算法的性能比

解顶点覆盖问题的一个启发式算法如下，每次选择具有最高度数的顶点，然后将与其关联的所有边删去。举例说明该算法的性能比将大于 2。

分析与解答：

对于 $n \geqslant 3, i \leqslant n$，定义 $A(n,i) = \sum\limits_{j=2}^{i} \left\lfloor \frac{n}{j} \right\rfloor$。构造图 $G_n = (V,E)$ 如下：

$$V = \{a_1, a_2, \cdots, a_{A(n,n-1)}, b_1, b_2, \cdots, b_n, c_1, c_2, \cdots, c_n\}$$

$$E = \{(b_i, c_i) \mid i = 1, \cdots, n\} \cup \bigcup_{i=2}^{n-1} \bigcup_{j=A(n,i-1)+1}^{A(n,i)} \{(a_j, b_k) \mid (j - A(n,i-1) - 1)i + 1 \leqslant k \leqslant (j - A(n,i-1))i\}$$

易见，$|V| = A(n,n-1) + 2n$，$nH(n-1) - n - (n-2) \leqslant A(n,n-1) \leqslant nH(n-1) - n$。

G_6 如图 8-1 所示。

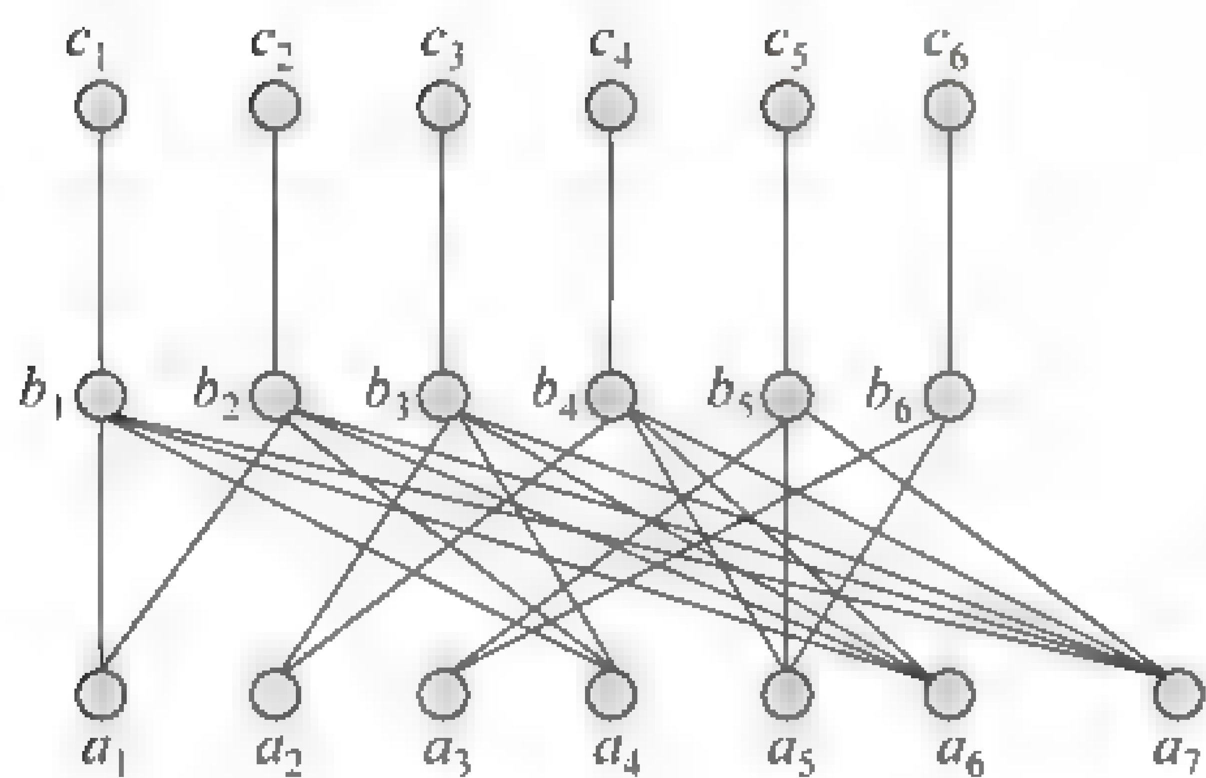


图 8-1　G_6 示意图

将顶点覆盖问题的启发式算法应用于图 $G_n = (V,E)$，选择顶点的序列为 $a_{A(n,n-1)}, a_{A(n,n-1)-1}, \cdots, a_1$；此后，剩下 n 条不相交的条边，必须再选 n 个顶点。因此，算法选出的覆盖顶点集中顶点数为 $A(n,n-1)+n$。而顶点集 $\{b_1, b_2, \cdots, b_n\}$ 显然是一个最优顶点覆盖，其顶点数为 n。

由此可见，启发式算法的性能比 η 满足 $\eta \geqslant (nH(n-1) - n + 2)/n \geqslant H(n-1) - 1$。

$\lim\limits_{n \to \infty} H(n-1) = \infty$，可见启发式算法的性能比 η 可任意大。

习题 8-10　团的常数性能比近似算法

图 G 的最优顶点覆盖是其补图中最大团集的补集。这个关系是否暗示对于团问题也有一个常数性能比的近似算法？

分析与解答：

设图 $G = (V,E)$，则 $S \subseteq V$ 是 G 的一个最大团当且仅当 $V - S$ 是 G 的一个最小顶点覆盖。而且团问题 CLIQUE 与顶点覆盖问题 VERTEX COVER 是多项式时间等价的。有人可能认为，对于等价的 NP 完全问题，可以用多项式变换将一个问题的好的近似算法转变为与之等价的另一个问题的好的近似算法。这种看法一般来说是不对的。事实上，若 G 是一

个有 1000 个顶点的图，而 G 的最小顶点覆盖的大小为 490。由于算法 approxVertexCover 的 η_1 值不超过 2，因此用它可以找到一个大小不超过 980 的顶点覆盖。但是，相应的团的比值可能大到

$$\eta_1 = \max\left(\frac{C^*}{C},\frac{C}{C^*}\right) = \frac{1000-490}{1000-980} = \frac{510}{20} = 25.5$$

由此可知，虽然多项式变换可以将一个问题的最优解变换为另一个问题的最优解，但不能保持最优解与近似最优解的比值不变。事实上，对于团问题 CLIQUE，可以证明不存在 η_1 值不依赖于 G 的规模的近似算法，除非 P=NP。

习题 8-11　售货员问题的常数性能比近似算法

证明旅行售货员问题的一个实例可在多项式时间内变换为该问题的另一个实例，使得其费用函数满足三角不等式，且两个实例具有相同的最优解。说明是否可以通过这个变换使得一般的旅行售货员问题具有一个常数性能比的近似算法。

分析与解答：

给定完全图 $G=(V,E)$，$|V|=n$，$|E|=\frac{n(n-1)}{2}=m$。设 (G,c) 是旅行售货员问题的一个实例，即 c 是 G 上的费用函数。若 c 不满足三角不等式，则可以在 $O(m)$ 时间内将旅行售货员问题的这个实例变换为它的另一个实例 (G,c_1)，使得 c' 满足三角不等式，且 (G,c_1) 与 (G,c) 有相同的最优解。也就是说，旅行售货员问题的一个实例可以在多项式时间内变换为该问题的另一个实例，使得其费用函数满足三角不等式，且两个实例具有相同的最优解。但这并不意味着通过所说的变换可使一般的旅行售货员问题有一个 η_1 是常数的多项式时间近似算法。与习题 8-10 的结论类似，虽然多项式变换可以将一个实例的最优解变换为另一个实例的最优解，但不能保证最优值与近似最优值的比值不变。事实上，在教材中已证明了在 P≠NP 的前提下，当费用函数不满足三角不等式时，对于任意的 $\rho \geqslant 1$，找不到解旅行售货员问题的多项式时间近似算法，使得该算法的 η_1 值以 ρ 为上界。

习题 8-12　瓶颈旅行售货员问题

瓶颈旅行售货员问题是要找出图 $G=(V,E)$ 的一条哈密顿回路，且使回路中最长边的长度最小。若费用函数满足三角不等式，给出解此问题的性能比为 3 的近似算法。（提示：递归地证明，可以通过对 G 的最小生成树进行完全遍历并跳过某些顶点，但不能跳过多于 2 个连续的中间顶点，以此方式访问最小生成树中每个顶点恰好一次）

分析与解答：

解瓶颈旅行售货员问题的性能比为 3 的近似算法描述如下：

```
void approxTSP (Graph G)
{
    ① 选择任一顶点 r∈V；
    ② 找出 G 的一棵以 r 为根的最小瓶颈生成树 T；
    ③ 选取 T 的一条边(p,q)作为哈密顿回路 H 的一条边；
    ④ 遍历树 T，跳过不多于两个连续的重复顶点，递归构造 H 的其他边；
    ⑤ 将所得到的哈密顿回路 H 作为计算结果返回。
}
```

设 $V=\{1,2,\cdots,n\}$。由习题4-12的结论可知,任何一棵最小生成树都是最小瓶颈生成树。因此可用Prim算法或Kruskal算法构造②的最小瓶颈生成树 T。事实上,可以在线性时间内构造最小瓶颈生成树。

下面讨论步骤①中,以最小瓶颈生成树 T 为基础,递归构造满足要求的哈密顿回路 H 的算法。

当 $n\leqslant 3$ 时,容易构造满足要求的哈密顿回路 H。当 $n>3$ 时:

(1) 将步骤③中选取的边 (p,q) 删去,树 T 分裂成两棵树 T_p 和 T_q,其中,T_p 包含顶点 p,而 T_q 包含顶点 q。

(2) 设 (p,p') 是树 T_p 中的一条边(如果存在);在树 T_p 中递归地构造以 (p,p') 为一条边的哈密顿回路 H_p。

(3) 设 (q,q') 是树 T_q 中的一条边(如果存在)。在树 T_q 中递归地构造以 (q,q') 为一条边的哈密顿回路 H_q。

(4) 删去 H_p 中的边 (p,p') 得到一条 $(p\rightarrow p')$ 哈密顿路 P_p;删去 H_q 中的边 (q,q') 得到一条 $(q'\rightarrow q)$ 哈密顿路 P_q;由此可得 q,p,P_p,q',P_q 是满足要求的以 (p,q) 为一条边的哈密顿回路,如图8-2所示。

由归纳假设知,H_p 和 H_q 中的边满足要求,即跳过不多于两个连续的 T 中顶点。按照上面的构造法,(p,q)、(p,p') 和 (q,q') 均为 T 的边。因此,如果边 (p',q') 不是 T 的边,它最多也只跳过 p 和 q 这两个顶点。由此可见,所构造的哈密顿回路是以 (p,q) 为一条边的满足要求的哈密顿回路。

下面讨论上述近似算法approxTSP的性能比。

设 G 的最小瓶颈生成树 T 中的最长边的长度为 c;G 最优瓶颈旅行售货员回路 H_{opt} 中的最长边的长度为 c_{opt};算法approxTSP构造出的哈密顿回路为 H,其最长边的长度为 c_H。从中任意删去一条边都构成 G 的1棵生成树,因此有 $c\leqslant c_{\text{opt}}$。设 (p,q) 为 H 的一条边。如果 (p,q) 不是 T 的边,由 H 的性质可知,(p,q) 与 T 中两条边构成一个三角形,或与 T 中三条边构成4边形,如图8-3所示。

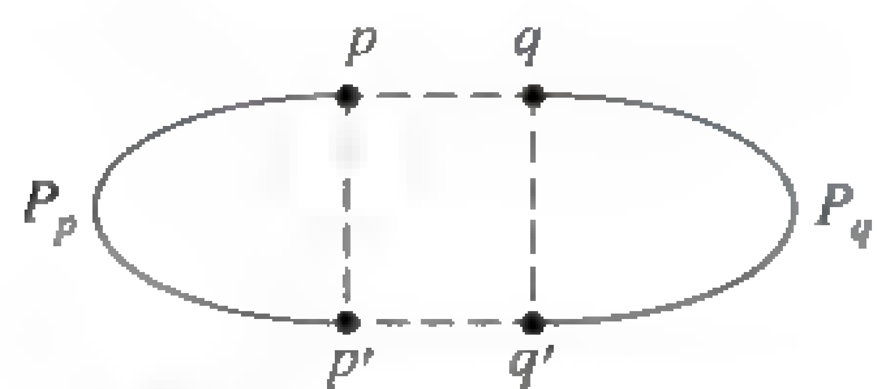


图8-2 构造哈密顿回路

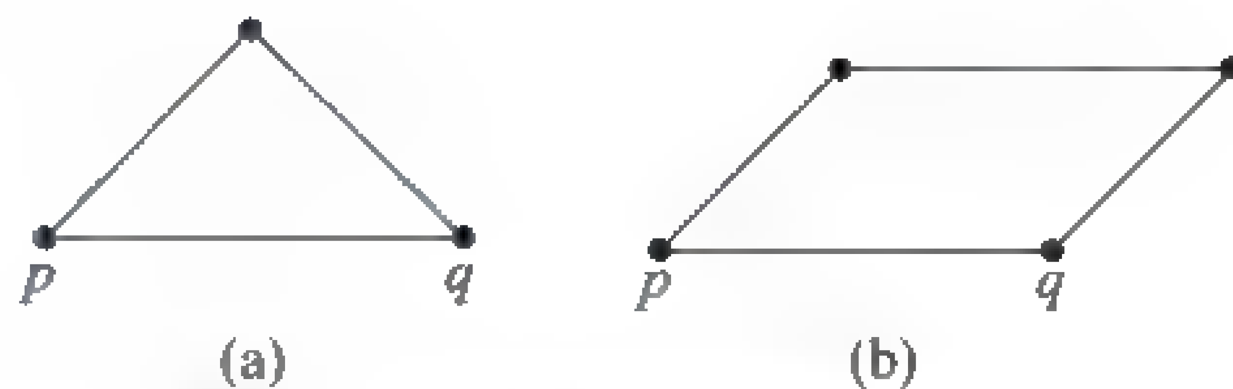


图8-3 哈密顿回路 H 中的边

由费用函数满足三角不等式可知,在图8-3(a)的情形有,$\text{dist}(p,q)\leqslant 2c$;在图8-3(b)的情形有,$\text{dist}(p,q)\leqslant 3c$。由此可推知,$c_H\leqslant 3c\leqslant 3c_{\text{opt}}$。也就是说,算法approxTSP的性能比为3。

习题8-13 最优旅行售货员回路不自相交

若旅行售货员问题中,图 G 的各顶点均为平面上的点,且费用函数 $c(u,v)$ 定义为点 u 和 v 之间的欧氏距离,证明 G 的最优旅行售货员回路不会自相交。

分析与解答:

对于所述的欧几里得旅行售货员问题,设 H 是 G 的一条最优旅行售货员回路,且 H 中

有2条边(u,v)和(s,t)相交于p。

由于H是哈密顿回路，故下面的两种情形必有一种情形会发生：

(1) $(u,s)\notin H$ 且 $(v,t)\notin H$。

(2) $(u,t)\notin H$ 且 $(v,s)\notin H$。

否则，在H中就会出现一个顶点关联3条以上的边，与H是哈密顿回路相矛盾。

由于情形(1)和情形(2)是对称的，不妨设情形(1)发生。此时，用边(u,s)和(v,t)代替边(u,v)和(s,t)得到$H=H\cup\{(u,s),(v,t)\}-\{(u,v),(s,t)\}$。易知$H$仍为$G$的一条哈密顿回路。由于$c(u,v)=|uv|=|up|+|pv|$，$c(s,t)=|st|=|sp|+|pt|$，且

$$|up|+|sp|>|us|=c(u,s),\quad |pv|+|pt|>|vt|=c(v,t)$$

故$c(u,v)+c(s,t)=|up|+|pv|+|sp|+|pt|>c(u,s)+c(v,t)$。

从而$c(H)>c(H)$，这与H是最优旅行售货员回路相矛盾。因此，对于欧几里得旅行售货员问题，任何一个最优旅行售货员回路不会自相交。

习题 8-14 集合覆盖问题的实例

试给出一族集合覆盖问题的实例，用以说明算法 greedySetCover 可以产生的不同解的个数随实例中元素个数n的指数增长。这里所说的不同解是指算法 greedySetCover 在作贪心选择时可以有多种选择，即使$|S\cap U|$最大的子集可有多个时，不同的选择导致算法的不同的解。

分析与解答：

先看一个简单实例。设$X=\{1,2,3,4\}$，$F=\bigcup S(i,j)$，其中，

$S(1,1)=\{1\}, S(1,2)=\{2\}, S(1,3)=\{3\}, S(1,4)=\{4\}$；

$S(2,1)=\{1,2\}, S(2,2)=\{1,3\}, S(2,3)=\{1,4\}$，

$S(2,4)=\{2,3\}, S(2,5)=\{2,4\}, S(2,6)=\{3,4\}$；

$S(3,1)=\{1,2,3\}, S(3,2)=\{1,2,4\}, S(3,3)=\{1,3,4\}, S(3,4)=\{2,3,4\}$。

算法 greedySetCover 第1次可选择$S(3,1)$，$S(3,2)$，$S(3,3)$，$S(3,4)$这4个集合中的任一集合。例如，选择集合$S(3,1)$后，F中剩余的集合为：$S(1,4)$，$S(2,3)$，$S(2,5)$，$S(2,6)$，$S(3,2)$，$S(3,3)$，$S(3,4)$。

算法 greedySetCover 接下来可选这7个集合中的任一集合。由此可见，对于这个实例，算法 greedySetCover 可产生28个不同的解。

在一般情况下，设$X=\{1,2,\cdots,n\}$，$F=\bigcup_{i=1}^{n-1}\bigcup_{j=1}^{\binom{n}{i}} S(i,j)$，其中，

$$S(1,1)=\{1\}, S(1,2)=\{2\},\cdots,S(1,n)=\{n\};$$

$$S(2,1)=\{1,2\}, S(2,2)=\{1,3\},\cdots,S\left(2,\binom{n}{2}\right)=\{n-1,n\};$$

$$\vdots$$

$$S(n-1,1)=\{1,2,\cdots,n-1\}, S(n-1,2)=\{1,2,\cdots,n-2,n\},\cdots,$$

$$S(n-1,n)=\{2,3,\cdots,n\}。$$

算法 greedySetCover 第1次可选择$S(n-1,1)$，$S(n-1,2)$，$\cdots$，$S(n-1,n)$这n个集合中的任一集合。例如，选择集合$S(n-1,1)$后，F中剩余的集合为

$S(1)$中$\binom{n-1}{0}$个集合;$S(2)$中$\binom{n-1}{1}$个集合…$S(n-1)$中$\binom{n-1}{n-2}$个集合。

总共有$\sum_{i=0}^{n-2}\binom{n-1}{i}=2^{n-1}-1$个集合。

算法 greedySetCover 接下来可选这 $2^{n-1}-1$ 个集合中的任一集合。由此可见,对于这类实例,算法 greedySetCover 可产生 $n(2^{n-1}-1)$个不同的解。

习题 8-15　多机调度问题的近似算法

多机调度问题:设有 m 台完全相同的机器完成 n 个彼此独立的任务,第 i 个任务所需的机器时间为 t_i,$i=1,2,\cdots,n$。要确定一个时间表,使全部 n 个任务都结束的时间最短。

解上述问题的最长处理时间算法 LPT 每次从待安排任务中选择最长处理时间的任务,并安排给一台完全空闲机器。试在 $O(n\log n)$时间内实现算法 LPT,并证明该算法所得到的解的相对误差$\lambda=\left|\frac{c^*-c}{c^*}\right|\leqslant\frac{1}{3}-\frac{1}{3m}$。

分析与解答:

算法描述见教材 1.7 节多机调度问题的解法。算法先将 n 个任务按照所需的机器时间从大到小排序。不失一般性,设 $t_1\leqslant t_2\leqslant\cdots\leqslant t_n$。对于多机调度问题的一个具体实例 x,设算法 LPT 计算出的完成时间为 $T_{\text{LPT}}(x)$,最优解的完成时间为 $T_{\text{opt}}(x)$。用 $s(i)=j$ 记录 LPT 算法将第 i 个任务分配给第 j 台机器。

下面讨论解的相对误差。

用反证法。设解的相对误差$\lambda=\left|\frac{c^*-c}{c^*}\right|>\frac{1}{3}-\frac{1}{3m}$,则存在多机调度问题的最小实例 I,使得$\left|\frac{T_{\text{LPT}}(I)-T_{\text{opt}}(I)}{T_{\text{LPT}}(I)}\right|>\frac{1}{3}-\frac{1}{3m}$。

由 I 是多机调度问题的最小实例,可推知任务 n 的完成时间是 $T_{\text{LPT}}(I)$。事实上,若任务 n 的完成时间不是 $T_{\text{LPT}}(I)$,则算法对任务集 $I'=\{t_1,t_2,\cdots,t_{n-1}\}$计算得到的完成时间也是 $T_{\text{LPT}}(I)$。又由于 I'是 I 的子集,故 $T_{\text{opt}}(I')\leqslant T_{\text{opt}}(I)$。由此可得

$$T_{\text{LPT}}(I')-T_{\text{opt}}(I')\geqslant T_{\text{LPT}}(I)-T_{\text{opt}}(I)\geqslant\left(\frac{1}{3}-\frac{1}{3m}\right)T_{\text{opt}}(I)\geqslant\left(\frac{1}{3}-\frac{1}{3m}\right)T_{\text{opt}}(I')$$

这与 I 是多机调度问题的满足$\left|\frac{T_{\text{LPT}}(I)-T_{\text{opt}}(I)}{T_{\text{LPT}}(I)}\right|>\frac{1}{3}-\frac{1}{3m}$的最小实例矛盾。

下面证明 $T_{\text{opt}}(I)<3t_n$。

设安排第 n 个任务前,m 台机器的状态为$\{T_1,T_2,\cdots,T_m\}$,$T_k=\min\{T_1,T_2,\cdots,T_m\}$。算法选择 $s(n)=k$。从前面的讨论可知,任务 n 的完成时间是 $T_{\text{LPT}}(I)$。因此,$T_k+t_n=T_{\text{LPT}}(I)$。

另一方面,$T_k=\min\{T_1,T_2,\cdots,T_m\}\leqslant\sum_{i=1}^{m}T_i/m$。由此可得

$$\sum_{i=1}^{n-1}t_i/m=\sum_{i=1}^{m}T_i/m\geqslant T_k=T_{\text{LPT}}(I)-t_n$$

因此,

$$\sum_{i=1}^{n} t_i \geqslant mT_{\text{LPT}}(I) - (m-1)t_n$$

由于每个任务都要在一台机器上完成，所以 $T_{\text{opt}}(I) \geqslant \sum_{i=1}^{n} t_i/m$。由此可得

$$\left(\frac{m-1}{m}\right)t_n \geqslant T_{\text{LPT}}(I) - T_{\text{opt}}(I) > \left(\frac{m-1}{3m}\right)T_{\text{opt}}(I), \quad T_{\text{opt}}(I) < 3t_n$$

而当 $T_{\text{opt}}(I)<3t_n$ 时，LPT 算法得到的是最优解。事实上，由 $T_{\text{opt}}(I)<3t_n$ 知，每台机器上的任务数不超过 2，否则 $T_{\text{opt}}(I)\geqslant 3t_n$。因此，$n\leqslant 2m$。不失一般性，可以假设 $n=2m$。因为如果 $n<2m$，可以加入 $2m-n$ 个任务使 $t_{n+1}=\cdots=t_{2m}=0$。

由算法 LPT 可知，任务 j 和任务 $2m-j+1$ 被安排在机器 j 上，$1\leqslant j\leqslant m$。设 $j=\max\{i \mid t_i+t_{2m-i+1}=T_{\text{LPT}}(I)\}$。另外设最优解将任务 i 分配给机器 $s_{\text{opt}}(i)$。构造顶点集为 $V=\{1,2,\cdots,n\}$的图 G 如下。当 $s(i)=s(k)$，即 $i+k=2m+1$ 时，加入蓝边(i,k)；当 $s_{\text{opt}}(i)=s_{\text{opt}}(k)$ 时，加入红边(i,k)。不论是最优解还是算法 LPT 的解，都在每台机器上恰好安排 2 个任务。因此红边和蓝边组成的图的每个顶点的度数恰为 2，从而图的每个连通分支都是一个简单圈。考查图中包含顶点 j 的连通分支。设它包含的顶点为$\{j_1,j_2,\cdots,j_l,2m-j_1+1,\cdots,2m-j_l+1\}$，$1\leqslant j_1,j_2,\cdots,j_l\leqslant m$。由于红边匹配了该圈中所有顶点，因此，存在红边$(i,k)$满足 $i\leqslant j, k\leqslant 2m-j+1$。

由 j 的定义知，$T_{\text{opt}}(I)\geqslant t_i+t_k\geqslant t_j+t_{2m-j+1}=T_{\text{LPT}}(I)$。

可见 $T_{\text{opt}}(I)=T_{\text{LPT}}(I)$，$\left|\frac{T_{\text{LPT}}(I)-T_{\text{opt}}(I)}{T_{\text{LPT}}(I)}\right|=0$。这与 I 的定义矛盾。从而证明了不存在多机调度问题的实例 I，使得 $\left|\frac{T_{\text{LPT}}(I)-T_{\text{opt}}(I)}{T_{\text{LPT}}(I)}\right|>\frac{1}{3}-\frac{1}{3m}$。换句话说，对于多机调度问题的任何实例 I，均有 $\left|\frac{T_{\text{LPT}}(I)-T_{\text{opt}}(I)}{T_{\text{LPT}}(I)}\right|\leqslant\frac{1}{3}-\frac{1}{3m}$，即算法所得到的解的相对误差 $\lambda=\left|\frac{c^*-c}{c^*}\right|\leqslant\frac{1}{3}-\frac{1}{3m}$。

习题 8-16　LPT 算法的最坏情况实例

设 $n=2m+1$ 且 $t_i=2m-\left\lfloor\frac{i+1}{2}\right\rfloor$，$1\leqslant i\leqslant 2m$，$t_{2m+1}=m$。试构造多机调度问题关于该实例的最优解 c^* 和用算法 LPT 求出的解 c，并计算近似算法 LPT 的性能比 $\eta=\left|\frac{c^*-c}{c^*}\right|$。

分析与解答：

对于所给实例，用近似算法 LPT 得到的解如下：

任务 i 和 $2m-i+1$ 安排在第 i 台机器，$1\leqslant i<m$；安排在第 m 台机器上的是任务 m，$m+1$ 和 $2m+1$。完成时间是 $t_m+t_{m+1}+t_{2m+1}=2m-\left\lfloor\frac{m+1}{2}\right\rfloor+2m-\left\lfloor\frac{m+2}{2}\right\rfloor+m=4m-1$。

所给实例的最优解是：任务 i 和 $2m-i-1$ 安排在第 i 台机器，$1\leqslant i<m$；安排在第 m 台机器上的是任务 $2m-1$，$2m$ 和 $2m+1$。完成时间是 $3m$。

由此可知，$\eta=\left|\frac{c^*-c}{c^*}\right|=\frac{4m-1-3m}{3m}=\frac{m-1}{3m}=\frac{1}{3}-\frac{1}{3m}$。

可见该实例使算法 LPT 达到解的相对误差的上界 $\frac{1}{3}-\frac{1}{3m}$。

习题 8-17　多机调度问题的多项式时间近似算法

设在多机调度问题中,要在所给 m 台机器上安排的 n 个任务已按各自所需处理时间的递减序列排列 $t_1 \geqslant t_2 \geqslant \cdots \geqslant t_n$。解此问题的算法 LPT2 先确定一个正整数 k,对前 k 个任务求最优安排,然后对后 $n-k$ 个任务用算法 LPT(习题 8-15)求解。

(1) 试证明算法 LPT2 的解的相对误差 $\lambda \leqslant \frac{1-1/m}{1+\lfloor k/m \rfloor}$。

(2) 根据(1)的结论,设计一个解多机调度问题的多项式时间近似算法,对于给定的 $\varepsilon > 0$,算法所需的计算时间为 $O(n\log n + m^{m/\varepsilon})$。

分析与解答:

(1) 设算法 LPT2 计算出的时间表长度为 T_{LPT2},最优时间表长度为 T_{opt},前 k 个任务的最优时间表长度为 t。如果 $T_{\mathrm{LPT2}}=t$,则命题成立。因此可设 $T_{\mathrm{LPT2}}>t$。设任务 j 的完成时间为 $T_{\mathrm{LPT2}}, j>k$。在时间 $T_{\mathrm{LPT2}}-t_j$ 处,所有机器非空闲。因此,$\sum_{i=1}^{j-1} t_i/m \geqslant T_{\mathrm{LPT2}}-t_j$。结合 $T_{\mathrm{opt}} \geqslant \sum_{i=1}^{j} t_i/m$ 可知,$T_{\mathrm{LPT2}}-T_{\mathrm{opt}} \leqslant \frac{m-1}{m}t_j \leqslant \frac{m-1}{m}t_{k+1}$。

由于 $t_i \geqslant t_{k+1}, 1 \leqslant i \leqslant k+1$,由鸽舍原理 m 台机器中,至少有一台机器上安排的任务数不少于 $1+\lfloor k/m \rfloor$。因此,$T_{\mathrm{opt}} \geqslant (1+\lfloor k/m \rfloor)t_{k+1}$。

结合前面的讨论即知,$\frac{T_{\mathrm{LPT2}}-T_{\mathrm{opt}}}{T_{\mathrm{opt}}} \leqslant \frac{(m-1)/m}{1+\lfloor k/m \rfloor} = \frac{1-1/m}{1+\lfloor k/m \rfloor}$。

(2) 对于给定的 $\varepsilon>0$,取 $k=\lfloor (m-1)/\varepsilon \rfloor$,可使 $\frac{1-1/m}{1+\lfloor k/m \rfloor}<\varepsilon$。

算法 LPT2 用 $O(m^k)$ 时间求前 k 个任务的最优时间表,算法其余部分所需的计算时间不超过 $O(n\log n)$。由此可见,算法 LPT2 是解多机调度问题的 ε 近似算法,所需的计算时间为 $O(n\log n + m^{m/\varepsilon})$。

算法实现题 8-1　旅行售货员问题的近似算法

★ 问题描述

教材中解旅行售货员问题的近似算法 approxTSP 可以进一步得到改进。由近似算法 $\eta=2$ 的证明过程容易看出,如果将 G 的最小生成树 T 的边看作是 G 的双重边,则回路 W 就是 T 的一个欧拉回路。而近似最优哈密顿回路是在这条欧拉回路中删除第 2 次经过的顶点得到的。如果基于 T 找出一条更短的欧拉回路,则可以得到一条更短的哈密顿回路。下面的算法框架就是基于这个思想来设计的。

```
void approxTSP(Graph G)
{
    ① 选择任一顶点 r∈V;
    ② 用 PRIM 算法找出 G 的一棵以 r 为根的最小生成树 T;
    ③ 找出 T 的奇数度顶点集 S;
    ④ 在以 S 为顶点集的 G 的完全子图中,找出一个最小完全匹配 M;
    ⑤ 在以 T 和 M 中所有边集组成的多重图中,找出一条欧拉回路;
    ⑥ 将找到的欧拉回路,除根 r 外第 2 次经过的顶点删去,得到一条哈密顿回路 H;
```

⑦ 将所得到的哈密顿回路 H 作为计算结果返回。

}

上述算法是解 TSP 问题的 $O(n^3)$ 时间近似算法，且其性能比达到 1.5。

★ **算法设计**

设计并实现上述近似算法。

★ **数据输入**

由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 e，n 表示 G 的顶点数；e 是 G 的边数。接下来的 e 行中，每行有 3 个正整数 i,j,c，表示边 (i,j) 的费用为 c。

★ **结果输出**

将近似最优哈密顿回路及其长度输出到文件 output.txt 中。文件的第 1 行是近似最优哈密顿回路的长度，第 2 行是近似最优哈密顿回路。

输入文件示例	输出文件示例
input.txt	output.txt
7 8	31
1 4 5	1 4 2 6 5 3 7
4 2 8	
2 6 3	
6 5 1	
5 3 3	
3 7 2	
7 1 9	
1 5 10	

分析与解答：

1) 算法正确性

算法的关键步骤是第④步，在 G 的奇数度顶点集 S 的完全子图中，找出一个最小完全匹配 M。设 $\deg(v)$ 是顶点 v 在树 T 中的度数，则 $2|T| = \sum_{v\in V}\deg(v) = \sum_{v\in S}\deg(v) + \sum_{v\in V-S}\deg(v)$。对任意 $v\in V-S$ 有，$\deg(v)$ 为偶数，故 $\sum_{v\in V-S}\deg(v)$ 为偶数，从而 $\sum_{v\in S}\deg(v)$ 也是偶数。又由于对任意 $v\in S$，$\deg(v)$ 为奇数，故 $|S|$ 为偶数。因此，第④步中的完全匹配 M 存在。图 $T+M$ 中各边的度数均为偶数，因此，$T+M$ 的欧拉回路也是存在的。由此可见算法的第⑤步可找到 $T+M$ 的欧拉回路，并在第⑥步中根据所找出的欧拉回路构造出 G 的一条哈密顿回路。

2) 算法的计算复杂性

算法的步骤②中 PRIM 算法需要 $O(n^2)$ 时间。步骤③显然只需要 $O(n)$ 时间。步骤④可在 $O(n^3)$ 时间内找出最小完全匹配 M。算法的步骤⑤和步骤⑥均可在 $O(n)$ 时间内完成。因此，整个算法所需的计算时间为 $O(n^3)$。

3) 算法的精度

设 G 的一个最优旅行售货员回路为：$H^*: v_1, v_2, \cdots, v_n, v_1$，并设 S 中的顶点为 v_{i_1}，

$v_{i_2},\cdots,v_{i_k},i_1<i_2<\cdots<i_k,k=|S|$。

由此可知，$M_1=\{(v_{i_{2j-1}},v_{i_{2j}})|1\leqslant j\leqslant k/2\}$和$M_2=\{(v_{i_{2j}},v_{i_{2j+1}})|1\leqslant j\leqslant k/2\}$是以$S$为顶点集的完全图2个完全匹配。因此，

$$C(M_1)+C(M_2)=\sum_{j=1}^{2k-1}C(v_{i_j},v_{i_{j+1}})+C(v_{i_{2k}},v_{i_1})\leqslant\sum_{j=1}^{n-1}C(v_j,v_{j+1})+C(v_n,v_1)$$

其中用到三角不等式性质，$C(v_{i_j},v_{i_{j+1}})\leqslant C(v_{i_j},v_{i_j+1})+\cdots+C(v_{i_{j+1}-1},v_{i_{j+1}})$。

因此，$C(M)\leqslant\min\{C(M_1),C(M_2)\}\leqslant C(H^*)/2$。

又由于$C(T)\leqslant C(H^*)$，故$C(T+M)=C(T)+C(M)\leqslant1.5C(H^*)$。

从而$C(H)\leqslant C(T+M)\leqslant1.5C(H^*)$。由此即知，所述算法的$\eta_t$值为1.5。

存在TSP问题的实例，使上述算法的近似最优值与最优值的比任意接近1.5。

算法实现题8-2　可满足问题的近似算法

★ 问题描述

设α是一个含有n个变量和m个合取项的合取范式。关于α的最大可满足性问题要求确定α的最多个数的合取式使这些合取式可同时满足。设k是α的所有合取式中因子个数的最小值。证明下面的解最大可满足问题的近似算法mSAT的相对误差为$\frac{1}{k-1}$。

```
Set mSAT(a)
{// xi,1≤i≤n,是a中n个变量;Ci,1≤i≤m,是α的m个合取项。
      cl=∅;
      left={Ci|1≤i≤m};
      lit={xi,xi|1≤i≤n};
      while (lit含有在left的合取式中出现的因子) {
          设y是lit的在left的合取式中出现次数最多的因子;
          设r是left中含有因子y的所有合取式的集合;
          cl=cl∪r;
          left=left-r;
          lit=lit-{y,y};
        }
      return (cl);
}
```

★ 算法设计

设计并实现上述近似算法。

★ 数据输入

由文件input.txt给出输入数据。第1行有2个正整数k和m，分别表示变量数和布尔表达式数。接下来的m行中，每行有若干个整数$i,j,k,\cdots,0$，表示表达式含的变量下标分别为$i,j,k,\cdots$，行末以0结尾。下标为负数时，表示相应的变量为取反变量。

★ 结果输出

将计算出的最大可满足合取式数输出到文件output.txt。

输入文件示例	输出文件示例
input. txt	output. txt
5 3	3
3 1 4 0	
1 −5 3 0	
2 −5 1 0	

分析与解答：

算法在题中已描述。算法所需的时间为 $O(n\log n)$。下面证明近似算法 mSAT 的相对误差为$\frac{1}{k+1}$。

设算法在 while 循环体内选中因子 y，r 是 left 中含有因子 y 的所有合取式的集合。left 中仅含因子 y 的合取项 b 未选中，但其所含的因子 y 从 lit 中删去，此时称合取项 b 被击中 1 次。留在 left 中的被击中的合取式的个数不超过选入 r 中合取式的个数。算法结束时仍留在 left 中的合取式的所有因子均已从 lit 中删除。这意味着算法结束时，至少有 $k|\text{left}|$ 次击中。因此算法结束时返回的可满足合取式的个数 $|\text{cl}|\geqslant k|\text{left}|$。因此，若最优值为 opt，则有

$$\text{opt}\leqslant m=|\text{cl}|+|\text{left}|\leqslant(1+1/k)|\text{cl}|$$

$$\frac{\text{opt}-|\text{cl}|}{\text{opt}}=1-\frac{|\text{cl}|}{\text{opt}}\leqslant 1-\frac{k}{k+1}=\frac{1}{k+1}$$

这个相对误差是可达的。例如，当 $k=3$ 时，设给定的合取范式为

$$\alpha=(\bar{x}_1+x_4+x_5)(\bar{x}_2+x_6+x_7)(\bar{x}_3+x_8+x_9)(x_1+x_2+x_3)$$

取 $x_1=x_4=x_6=x_8=\text{true}$，可使 4 个合取式都满足，因此，opt＝4。而算法 mSAT 在选取 $x_1=x_2=x_3=\text{true}$ 后，使合取式$(x_1+x_2+x_3)$不满足。可见，$\frac{\text{opt}-|\text{cl}|}{\text{opt}}=\frac{4-3}{4}=\frac{1}{4}$。

算法实现题 8-3　最大可满足问题的近似算法

★ 问题描述

证明下面的解最大可满足问题的近似算法 mSAT2 的相对误差为$\frac{1}{2^k}$，k 是 α 的所有合取式中因子个数的最小值。

```
Set mSAT2(a)
{// x_i,1≤i≤n,是 a 中 n 个变量;c_i,1≤i≤m,是 a 的 m 个合取项。
    for (int i=1;i<=m;i++) w[i]=2^(-|c_i|);
    cl=∅;
    left={c_i|1≤i≤m};
    lit={x_i,x_i|1≤i≤n};
    while (lit 含有在 left 的合取式中出现的因子) {
        设 y 是 lit 的在 left 的合取式中出现的因子;
        设 r 是 left 中含有因子 y 的所有合取式的集合;
        设 s 是 left 中含有因子 y 的所有合取式的集合;
        if ( Σ_{c_i∈r} w[i] ≥ Σ_{c_i∈s} w[i] ) {
```

```
            cl=cl∪r;left=left-r;
            对所有 c_i∈s,w[i]=2*w[i];}
        else {cl=cl∪s;left=left-s;
            对所有 c_i∈r,w[i]=2*w[i];}
        lit=lit-{y,ȳ};}
    return (cl);
}
```

★ **算法设计**

设计并实现上述近似算法。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m，分别表示变量数和布尔表达式数。接下来的 m 行中，每行有若干个整数 $i,j,k,\cdots,0$，表示表达式含的变量下标分别为 $i,j,k,\cdots$，行末以 0 结尾。下标为负数时，表示相应的变量为取反变量。

★ **结果输出**

将计算出的最大可满足合取式数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 3	3
3 1 4 0	
1 -5 3 0	
2 -5 1 0	

分析与解答：

算法在题中已描述。算法所需的时间为 $O(n\log n)$。下面证明近似算法 mSAT2 的相对误差为$\frac{1}{2^k}$。

初始时，$\sum_{i=1}^{m} w[i] \leqslant m/2^k$。在算法的 while 循环体内，每次迭代留在 left 中的被击中的合取式增加的权值不超过选入 r 中合取式的权值。因此，left 中合取式的总权值不增加，从而在算法结束时，left 中合取式的总权值不超过 $m/2^k$；另一方面，在算法结束时，留在 left 中的每个合取式的权值为 1。由此可知，在算法结束时，$|\text{left}| \leqslant m/2^k$。因此，算法结束时返回的可满足合取式的个数 $|\text{cl}| = m - |\text{left}| \geqslant m(1-1/2^k)$。

若最优值为 opt，则有

$$\text{opt} \leqslant m \leqslant \frac{2^k}{2^k-1}|\text{cl}|$$

$$\frac{\text{opt}-|\text{cl}|}{\text{opt}} = 1 - \frac{|\text{cl}|}{\text{opt}} \leqslant 1 - \frac{2^k-1}{2^k} = \frac{1}{2^k}$$

这个相对误差是可达的。例如，当 $k=3$ 时，设给定的合取范式为

$$\alpha = (x_1+x_4+x_5)(x_1+x_6+x_7)(x_1+x_8+x_9)(x_1+x_{10}+x_{11})$$
$$(x_1+x_2+x_3)(x_1+x_2+x_3)(x_1+x_2+x_3)(x_1+x_2+x_3)$$

取 $x_1=x_4=x_6=x_8=x_{10}=\text{true}$，可使 8 个合取式都满足，因此，opt $=8$。算法 mSAT2 在选

取 $x_1=\text{true}$ 后，使含有 x_1 的 4 个合取式之一不满足。可见，$\frac{\text{opt}-|\text{cl}|}{\text{opt}}=\frac{8-7}{8}=\frac{1}{8}$。

算法实现题 8-4　子集和问题的近似算法

★ 问题描述

子集和问题的一个实例为〈S,t〉。其中，$S=\{x_1,x_2,\cdots,x_n\}$ 是一个正整数的集合，t 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1，使得 $\sum\limits_{x\in S_1}x=t$。

在实际应用中，常遇到最优化形式的子集和问题。在这种情况下，要找出 S 的一个子集 S_1，使得其和不超过 t，又尽可能地接近 t。

★ 算法设计

对于给定的子集和问题的一个实例〈S,t〉，设计一个算法找出 S 的一个子集 S_1，使得其和不超过 t，又尽可能地接近 t。

★ 数据输入

由文件 input. txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 t，n 表示 S 的大小，t 是子集和的目标值。第 2 行中有 n 个正整数，表示集合 S 中的元素。

★ 结果输出

将子集和的最优值输出到文件 output. txt 中。

输入文件示例	输出文件示例
input. txt	output. txt
3 7	6
1 4 5	

分析与解答：

对教材中算法作适当修改。

算法实现题 8-5　子集和问题的完全多项式时间近似算法

★ 问题描述

子集和问题的一个实例为〈S,t〉。其中，$S=\{x_1,x_2,\cdots,x_n\}$ 是一个正整数的集合，t 是一个正整数。子集和问题判定是否存在 S 的一个子集 S_1，使得 $\sum\limits_{x\in S_1}x=t$。

在实际应用中，常遇到最优化形式的子集和问题。在这种情况下，要找出 S 的一个子集 S_1，使得其和不超过 t，又尽可能地接近 t。

★ 算法设计

对于给定的子集和问题的一个实例〈S,t〉，设计一个完全多项式时间近似算法找出 S 的一个子集 S_1，使得其和不超过 t，又尽可能地接近 t。

★ 数据输入

由文件 input. txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 t，n 表示 S 的大小，t 是子集和的目标值。第 2 行中有 n 个正整数，表示集合 S 中的元素。

★ 结果输出

将子集和的最优值输出到文件 output. txt 中。文件的第 1 行是 n 和 t 的值。第 2 行是计算出的近似最优值。

输入文件示例	输出文件示例
input. txt	output. txt
17 100	17 100
10 8 8 5 5 6 3 6 2 9 2 10	100
10 4 9 3 6	

分析与解答：

对教材中算法作适当修改。

算法实现题 8-6　实现算法 greedySetCover

★ 问题描述

集合覆盖问题的一个实例$\langle X,F\rangle$由一个有限集 X 及 X 的一个子集族 F 组成。子集族 F 覆盖了有限集 X。也就是说 X 中每一元素至少属于 F 中的一个子集，即 $X=\bigcup_{S\in F}S$。对于 F 中的一个子集 $C\subseteq F$，若 C 中的 X 的子集覆盖了 X，即 $X=\bigcup_{S\in C}S$，则称 C 覆盖了 X。集合覆盖问题就是要找出 F 中覆盖 X 的最小子集 C^*，使得 $|C^*|=\min\{|C| \mid C\subseteq F$ 且 C 覆盖 $X\}$。

设计并实现算法 greedySetCover，使其计算时间为 $O\left(\sum_{s\in F}|s|\right)$。

★ 算法设计

实现集合覆盖问题的近似算法 greedySetCover。

★ 数据输入

由文件 input. txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，分别表示有限集 X 中元素个数和子集族 F 中子集个数。$X=\{0,1,\cdots,n-1\}$，$F=\{f_0,f_1,\cdots,f_{m-1}\}$。接下来的 m 行中，每行对应于 F 中一个子集 f_i。第一个数是子集 f_i 中元素个数 k_i，接着的 k_i 个数表示 f_i 中的元素。

★ 结果输出

将计算出的最小覆盖子集输出到文件 output. txt。第 2 行是最小覆盖子集数。

输入文件示例	输出文件示例
input. txt	output. txt
12 6	4
6 0 1 2 3 4 5	0421
4 0 3 6 9	
4 1 4 7 10	
4 4 5 7 8	
4 2 5 8 11	
2 9 10	

分析与解答：

设给定的有限集为 $X=\{0,1,\cdots,n-1\}$；X 的子集族为 $F=\{f_0,f_1,\cdots,f_{m-1}\}$。

建立集合表 F，顶点表 V 和集合秩表 R 如下。

$F[i]$存储集合 f_i 中的元素，$0 \leq i < m$；$V[i]$存储包含元素 i 的集合，$0 \leq i < n$；$R[i]$存储当前集合大小为 i 的集合，$0 < i \leq n$。用双链表存储 $R[i]$中的集合。

(1) 算法从 $R[n]$开始，扫描集合秩表 R，取出当前剩余元素最多的集合 $F[i]$。

(2) 对于 $F[i]$中每个元素 j，根据 $V[j]$存储的每个集合 k，将 $F[k]$中的元素 j 删去，并修改集合 $F[k]$在 R 中的位置。

具体算法描述如下。

用类 fType 表示 F 中元素信息。

```
static class fType
{
    int size,rank;
    int []E;
    DoubleNode p;
    fType(int sz,int rk,int []ee,DoubleNode pp)
    {size=sz;rank=rk;E=ee;p=pp;}
    fType() {size=0;rank=0;E=new int[MAXELE];}
}
```

其中，size 表示相应集合中元素个数；rank 表示当前集合中元素个数；p 是指向集合在 R 的双链表中元素的指针，用于实现 R 的双链表中元素的 $O(1)$时间删除运算。数组 E 存放集合中的元素。

用类 vType 表示 V 中元素信息。

```
static class vType
{
    int []E;
    int size;
    vType(int sz,int []ee) {size=sz;E=ee;}
    vType() {size=0;E=new int[MAXELE];}
}
```

其中，size 表示包含相应元素的集合个数；数组 E 存放相应集合。

下面是算法中用到的变量。

```
static int n,m,cn=0;
static int []C=new int[MAXSET];
static int []U=new int[MAXELE];
static int []FU=new int[MAXSET];
static DoublyLinkedList []R;
static vType []V=new vType [MAXELE];
static fType []F=new fType [MAXSET];
```

变量 n 表示集合 X 中元素个数。变量 m 表示子集族 F 中集合个数。C[MAXSET]存储解集，cn 是解集 C 中集合个数。U[MAXELE]存储当前未删除元素。FU[MAXSET]存储当前未选择集合。数组 R 存储集合秩表，$R[i]$存储当前集合大小为 i 的集合组成的双链表。数组 F 存储集合中的元素。数组 V 存储包含元素的集合信息。

init 读入初始数据,并建立表 F,V 和 R,初始化 U 和 FU。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    m=keyboard.readInt();
    R=new DoublyLinkedList[n+1];
    for(int i=0;i<=n;i++) R[i]=new DoublyLinkedList();
    for(int i=0;i<n;i++){V[i]=new vType();F[i]=new fType();}
    for(int i=0;i<n;i++){U[i]=1;V[i].size=0;}
    for(int i=0;i<m;i++){
        FU[i]=1;
        F[i].size=keyboard.readInt();
        F[i].rank=F[i].size;
            for(int k=0,j=0;k<F[i].size;k++){
            j=keyboard.readInt();
            F[i].E[k]=j;
            V[j].E[V[j].size++]=i;
        }
    }
    for(int i=0;i<m;i++){
        R[F[i].rank].add(0,new Integer(i));
        F[i].p=R[F[i].rank].first();
    }
}
```

算法主体 greedySetCover 描述如下:

```
static boolean greedySetCover()
{
    int i,j,t,ei,si,ti,fi,vi,k=n,total=n;
    while(total>0 && k>0){
        if(! R[k].isEmpty()){
            si=((Integer)R[k].remove (0)).intValue();
            FU[si]=0;
            total-=k;
            C[cn++]=si;                              // 将集合 si 加入解集 C
            // 将 F[si]中的每个元素从 F 中其他集合中删除
            for(i=0,t=F[si].size;i<t;i++){
                vi=F[si].E[i];                       // F[si]中的元素
                if(U[vi]>0){
                    for(j=0,ei=V[vi].size;j<ei;j++){
                        ti=V[vi].E[j];               // 包含元素 vi 的集合
                        if(FU[ti]>0){
                            fi=F[ti].rank;
```

```
                        R[fi].remove (F[ti].p);              // 用 O(1)时间删除 F[ti]
                        if(fi>1){
                            R[fi-1].add(0,new Integer(ti));  // 用 O(1)时间插入 F[ti]
                            F[ti].p=R[fi-1].first();         // 保存指向 F[ti]的指针
                            F[ti].rank--;
                        }
                    }
                }
                U[vi]=0;
            }
        }
    }
    else k--;
  }
  return total==0;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
    init();
    if(greedySetCover())out();
    else System.out.println("No Solution!");
}
```

out 输出计算结果。

```
static void out()
{
    System.out.println(cn);
    for(int i=0;i<cn;i++)System.out.print(C[i]+" ");
    System.out.println();
}
```

算法主体 greedySetCover 的关键之处是可以用 $O(1)$时间将集合从 R 的双链表中删除；将集合插入 R 的双链表中也只要 $O(1)$时间。在最坏情况下，算法访问 F 的集合中每个元素 1 次，每次耗费 $O(1)$时间。因此，在最坏情况下算法计算时间为 $O\left(\sum_{s\in F}|s|\right)$。

算法实现题 8-7　装箱问题的近似算法 First Fit

★ 问题描述

设有体积分别为 $v_1, v_2, \cdots, v_n$ 的 n 种物品要装入容量为 c 的箱子里。不同的装箱方案所需的箱子数可能不同。装箱问题要求找出一种装完这 n 种物品所需的箱子数最少的装箱方案。装箱问题的近似算法 First Fit 的基本思想是，n 种物品依次装箱，将物品 i 装入第 1 个能装下它的箱子里。即物品 i 被装入已装入物品的体积不超过 $c-v_i$ 的编号最小的箱子里。

★ **算法设计**

设计并实现装箱问题的近似算法 First Fit。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有2个正整数 n 和 c，分别表示有 n 种物品要装入容量为 c 的箱子里。第2行中有 n 个正整数，分别表示 n 种物品的体积。

★ **结果输出**

将计算出的最少箱子数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	6
3 4 4 3 5 1 2 5 3 1	

分析与解答：

(1) 首先证明装箱问题是 NP 难的。

事实上，可以将划分问题变换为装箱问题。设 $\{a_1, a_2, \cdots, a_n\}$ 是划分问题的一个实例。变换为装箱问题的实例如下：$v_i = a_i, i = 1, \cdots, n$；$c = \sum_{i=1}^{n} a_i / 2$。显然，最少箱子数为2的充要条件是存在 $\{a_1, a_2, \cdots, a_n\}$ 的一个划分。

(2) 用竞赛树实现 First Fit 策略。

竞赛树结点 Bin 定义如下：

```
public static class Bin implements Playable
{
    int unc;
    public Bin(int uncc){unc=uncc;}
    public boolean winnerOf(Playable binn)
    {
        if (unc>=((Bin) binn).unc) return true;
        else return false;
    }
}
```

firstFitPack 实现 First Fit 搜索。

```
public static int firstFitPack(int []sz, int binc)
{
    int nn=0,n=sz.length-1;
    Bin []bin=new Bin [n+1];
    WinnerTree winTree=new WinnerTree();
    // 初始化 n 个箱子对应的竞赛树结点
    for (int i=1;i<=n;i++)bin[i]=new Bin(binc);
    winTree.initialize(bin);
    // 装箱
    for (int i=1;i<=n;i++){
```

```
        int child=2;                            // 开始搜索
        while (child<n){
            int winner=winTree.getWinner(child);
            if (bin[winner].unc<sz[i]) child++;     // 在右子树中
            child *=2;                          // 进入左子树
        }
        int binu=0;                             // 要选择的箱子号
        child/=2;
        if (child<n){
            binu=winTree.getWinner(child);
            // 如果 binu 是右儿子结点则须考查第 binu-1 号箱子
            if (binu>1 && bin[binu-1].unc>=sz[i]) binu--;
        }
        else binu=winTree.getWinner(child/2);
        if(bin[binu].unc==binc && binu>nn)nn=binu;
        bin[binu].unc-=sz[i];
        winTree.rePlay(binu);
    }
    return nn;
}
```

实现算法的主函数如下：

```
public static void main(String []args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int binc=keyboard.readInt();
    int []sz=new int [n+1];
    for (int i=1;i<=n;i++)sz[i]=keyboard.readInt();
    System.out.println(firstFitPack(sz, binc));
}
```

算法实现题 8-8　装箱问题的近似算法 Best Fit

★ 问题描述

设有体积分别为 $v_1, v_2, \cdots, v_n$ 的 n 种物品要装入容量为 c 的箱子里。不同的装箱方案所需的箱子数可能不同。装箱问题要求找出一种装完这 n 种物品所需的箱子数最少的装箱方案。装箱问题的近似算法 Best Fit 的基本思想是，n 种物品依次装箱，将物品 i 装入箱子 j 应满足 $c-c_j-v_i=\min\limits_{c-c_k-v_i\geq 0}\{c-c_k-v_i\}$，即选取第 j 号箱子，使得装入物品 i 后所留空隙最小。其中，c_k 表示已装入第 k 号箱子的物品的体积。

★ 算法设计

设计并实现装箱问题的近似算法 Best Fit。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c，分别表示有 n 种物品要

装入容量为c的箱子里。第2行中有n个正整数,分别表示n种物品的体积。

★ **结果输出**

将计算出的最少箱子数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	6
3 4 4 3 5 1 2 5 3 1	

分析与解答:

用二叉搜索树实现 Best Fit 策略。

二叉搜索树结点 BinNode 定义如下:

```
public static class BinNode
{
    int id, unc;
    public BinNode(int idd, int uncc){id=idd;unc=uncc;}
}
```

bestFitPack 实现 Best Fit 搜索。

```
public static int bestFitPack(int []sz, int binc)
{
    int nn=0,n=sz.length-1;
    int binu=0;
    BSearchTree stree;
    stree=new BSearchTree();
    // 装箱
    for (int i=1; i<=n; i++){
      // 找最合适的箱子
      BinNode bestBin=(BinNode) stree.getGE(new Integer(sz[i]));
      if (bestBin==null)                                    // 启用新箱子
        bestBin=new BinNode(++binu, binc);
      else bestBin=(BinNode) stree.remove(new Integer(bestBin.unc));
      // 修改箱子容量
      if(bestBin.unc==binc)nn=bestBin.id;
      bestBin.unc-=sz[i];
      if (bestBin.unc>0)stree.put(new Integer(bestBin.unc), bestBin);
    }
    return nn;
}
```

实现算法的主函数如下:

```
public static void main(String [ ]args)
{
    ReadStream keyboard=new ReadStream();
```

```
        int n=keyboard.readInt();
        int binc=keyboard.readInt();
        int [ ]sz=new int [n+1];
        for (int i=1;i<=n;i++)sz[i]=keyboard.readInt();
        System.out.println(bestFitPack(sz, binc));
}
```

算法实现题 8-9　装箱问题的近似算法 First Fit Decreasing

★ **问题描述**

设有体积分别为 $v_1, v_2, \cdots, v_n$ 的 n 种物品要装入容量为 c 的箱子里。不同的装箱方案所需的箱子数可能不同。装箱问题要求找出一种装完这 n 种物品所需的箱子数最少的装箱方案。装箱问题的近似算法 First Fit Decreasing 的基本思想是,先将 n 个物品依其体积从大到小排序 $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_n$,然后用算法 First Fit 求解。

★ **算法设计**

设计并实现装箱问题的近似算法 First Fit Decreasing。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c,分别表示有 n 种物品要装入容量为 c 的箱子里。第 2 行中有 n 个正整数,分别表示 n 种物品的体积。

★ **结果输出**

将计算出的最少箱子数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	6
3 4 4 3 5 1 2 5 3 1	

分析与解答:

先将 n 个物品依其体积从大到小排序,然后用 firstFitPack 求解。

算法实现题 8-10　装箱问题的近似算法 Best Fit Decreasing

★ **问题描述**

设有体积分别为 $v_1, v_2, \cdots, v_n$ 的 n 种物品要装入容量为 c 的箱子里。不同的装箱方案所需的箱子数可能不同。装箱问题要求找出一种装完这 n 种物品所需的箱子数最少的装箱方案。装箱问题的近似算法 Best Fit Decreasing 的基本思想是,先将 n 个物品依其体积从大到小排序 $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_n$,然后用算法 Best Fit 求解。

★ **算法设计**

设计并实现装箱问题的近似算法 Best Fit Decreasing。

★ **数据输入**

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c,分别表示有 n 种物品要装入容量为 c 的箱子里。第 2 行中有 n 个正整数,分别表示 n 种物品的体积。

★ **结果输出**

将计算出的最少箱子数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	6
3 4 4 3 5 1 2 5 3 1	

分析与解答：

先将 n 个物品依其体积从大到小排序，然后用 bestFitPack 求解。

算法实现题 8-11　装箱问题的近似算法 Next Fit

★ 问题描述

设有体积分别为 $v_1, v_2, \cdots, v_n$ 的 n 种物品要装入容量为 c 的箱子里。不同的装箱方案所需的箱子数可能不同。装箱问题要求找出一种装完这 n 种物品所需的箱子数最少的装箱方案。装箱问题的近似算法 Next Fit 的基本思想是，n 种物品依次装箱，将物品 i 装入下一个能装下它的箱子里，即从最近用过的箱子开始找下一个能装下物品 i 的箱子 k，将物品 i 装入箱子 k。

★ 算法设计

设计并实现装箱问题的近似算法 Next Fit。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c，分别表示有 n 种物品要装入容量为 c 的箱子里。第 2 行中有 n 个正整数，分别表示 n 种物品的体积。

★ 结果输出

将计算出的最少箱子数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	6
3 4 4 3 5 1 2 5 3 1	

分析与解答：

用竞赛树实现 Next Fit 策略。

竞赛树结点存储箱子的容量。

nextFitPack 实现 Next Fit 搜索。

```
public static int nextFitPack(int s[], int c)
{
    int nn=0,n=s.length-1;
    FirstFit.Bin []bin=new FirstFit.Bin [n+1];
    WinnerTree winTree=new WinnerTree();
    // 初始化 n 个箱子对应的竞赛树结点
    for (int i=1; i<=n; i++) bin[i]=new FirstFit.Bin(c);
    winTree.initialize(bin);
    int last=0;                              // 最近使用的箱子号
    // 装箱
    for (int i=1; i<=n; i++){                // 开始搜索
```

```
int j=last+1;
if (bin[j].unc<s[i])
    if (bin[j+1].unc>=s[i]) j++;
    else{                                   // 置 p 为箱子 j 的父结点
        int p=winTree.parent(j);
        boolean done=false;
        if (p==n-1){                        // 特殊情况
            int q;
            if (j%2==1) q=j+1;
            else q=j+2;
            // q <=n
            if (bin[q].unc>=s[i]){j=q;done=true;}
        }
        if (!done){                         // p 的够容量最近祖先结点
            p/=2;
            while (bin[winTree.getWinner(p)].unc<s[i]) p/=2;
            // p 的够容量最左子孙结点
            p*=2;
            while (p<n){
                int winp=winTree.getWinner(p);
                if (bin[winp].unc<s[i])p++;            // 在右子树中
                p*=2;
            }
            p/=2;
            if (p<n){
                j=winTree.getWinner(p);
                // 如果 j 是右儿子结点则须考查第 j-1 号箱子
                if (j>1 && bin[j-1].unc>=s[i]) j--;
            }
            else j=winTree.getWinner(p/2);             // n 为奇数
        }
    }
// 箱子 j 是否非空
if (bin[j].unc==c){
    // 够容量最左箱子
    int p=2;
    while (p<n){
        int winp=winTree.getWinner(p);
        if (bin[winp].unc<s[i])p++;                    // 在右子树中
        p*=2;
    }
    p/=2;
    if (p<n){
        j=winTree.getWinner(p);
        // 如果 j 是右儿子结点则须考查第 j-1 号箱子
```

```
                if (j>1 && bin[j-1].unc>=s[i]) j--;
            }
            else j=winTree.getWinner(p/2);                    // n为奇数
        }
        if(bin[j].unc==c && j>nn)nn=j;
        bin[j].unc -= s[i];
        winTree.rePlay(j);
        last=j;
    }
    return nn;
}
```

实现算法的主函数如下：

```
public static void main(String []args)
{
    ReadStream keyboard=new ReadStream();
    int n=keyboard.readInt();
    int binc=keyboard.readInt();
    int []sz=new int [n+1];
    for (int i=1;i<=n;i++)sz[i]=keyboard.readInt();
    System.out.println(nextFitPack(sz, binc));
}
```

第 9 章 串与序列的算法

习题 9-1　简单子串搜索算法最坏情况复杂性

试说明简单子串搜索算法在最坏情况下的计算时间复杂性为 $\Theta(m(n-m+1))$。

分析与解答：

考查一个特例。

设 $t=a^n$（即由连续 n 个 a 组成的串），$p=a^{m-1}b$。

在简单子串搜索算法 search 中，对于第 1 个循环的每个 i，都需要对 p 做 m 次比较。因此，总比较次数为 $m(n-m+1)$。由此可见，简单子串搜索算法在最坏情况下的计算时间复杂性为 $\Theta(m(n-m+1))$。当 $m=n/2$ 时，所需计算时间为 $\Theta(n^2)$。

习题 9-2　后缀重叠问题

设 x，y 和 z 是 3 个串，且满足 $x\quad z$ 和 $y\quad z$。试证明：

(1) 若 $|x|\leqslant|y|$，则 $x\quad y$。

(2) 若 $|x|\geqslant|y|$，则 $y\quad x$。

(3) 若 $|x|=|y|$，则 $x=y$。

分析与解答：

因为(2)与(1)是对称的，所以只要将图 9-1(a)中的 x 和 y 互换即可。从图 9-1(a)容易看出结论(1)和(2)的正确性；从图 9-1(b)容易看出结论(3)的正确性。

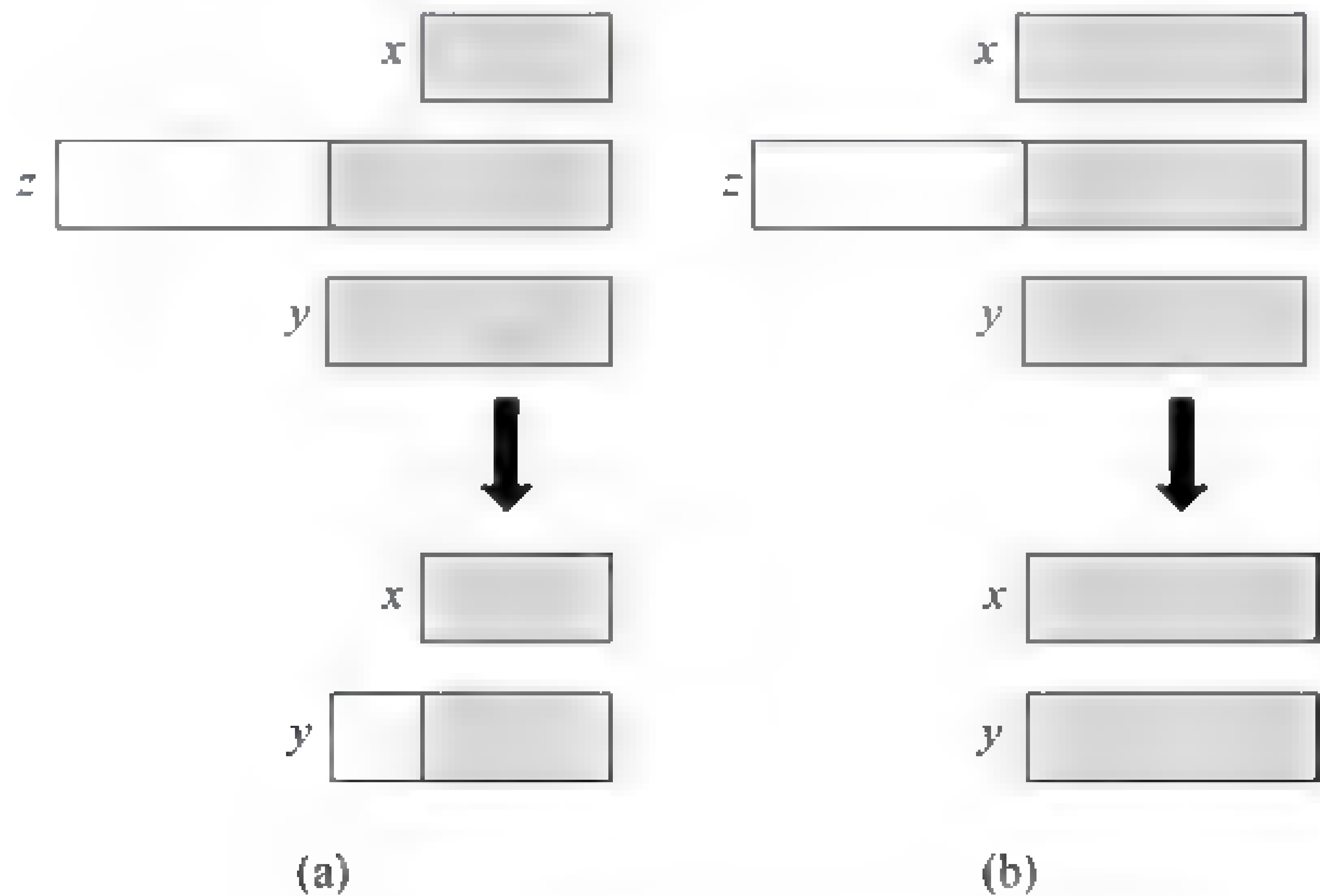


图 9-1　后缀重叠问题

习题 9-3　改进前缀函数

KMP 算法通过模式串的前缀函数，较好地利用了搜索过程中的部分匹配信息，从而提高了效率。然而在某些情况下，还可以更好地利用部分匹配信息。例如，考查图 9-2 中，KMP 算法对主串 aaabaaaab 和模式串 aaaab 的搜索过程。

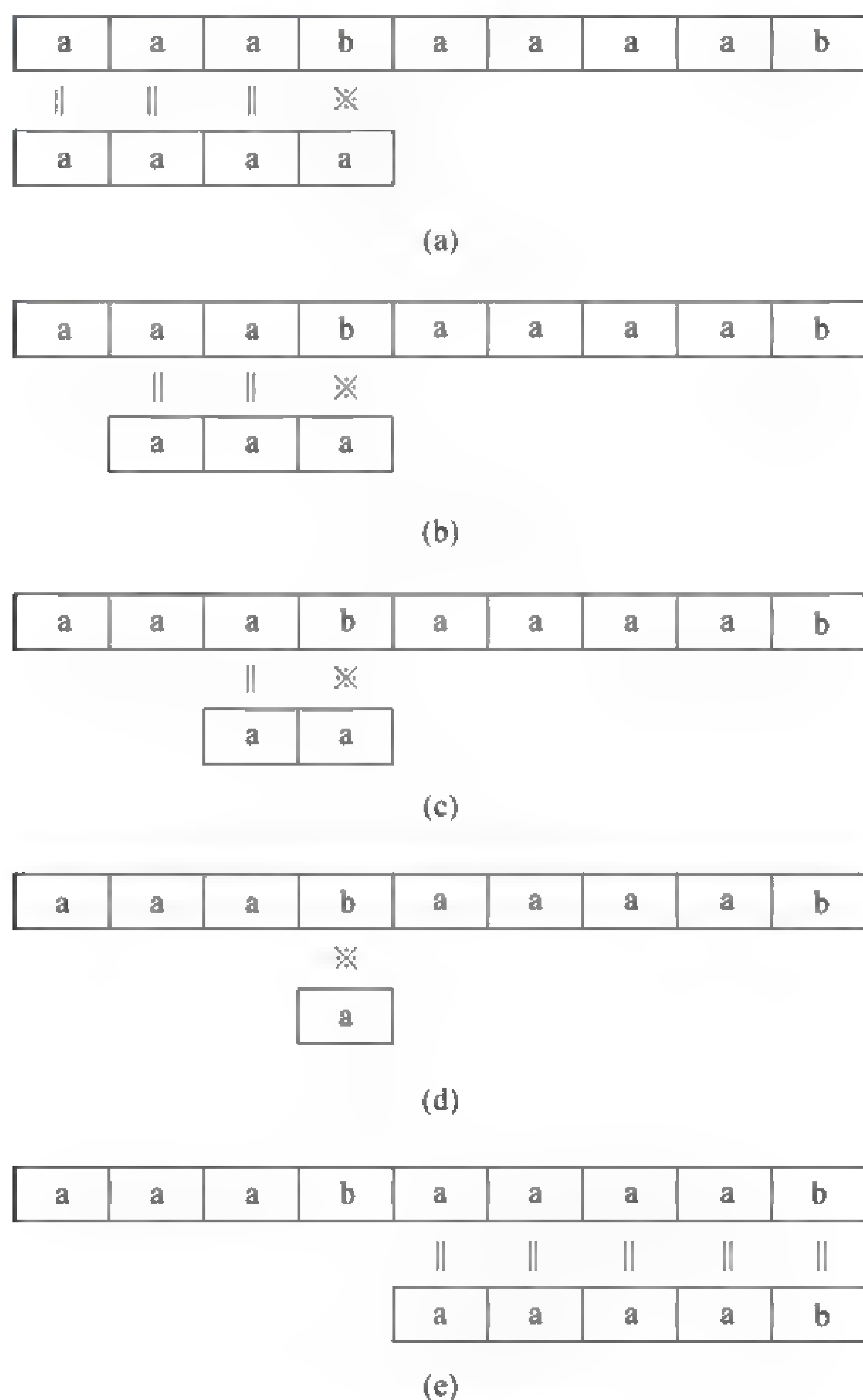


图 9-2　改进前缀函数

在图 9-2(a)中匹配失败后，按前缀函数指示继续做图 9-2(b)至图 9-2(d)的比较后，最后在图 9-2(e)找到一个匹配。事实上图 9-2(b)～图 9-2(d)的比较都是多余的。因为模式串在位置 0，1，2 处的字符和位置 3 处的字符都相等，因此不需要再和主串中位置 3 处的字符比较，而可以将模式一次向右滑动 4 个字符，直接进入图 9-2(e)的比较。这就是说，在 KMP 算法中遇到 $p[j+1]\neq t[i]$，且 $p[j+1]=p[\text{next}[j]+1]$时，可一次向右滑动 $j-\text{next}[\text{next}[j]]$个字符，而不是 $j-\text{next}[j]$个字符。根据此观察，设计一个改进的前缀函数，使得遇到上述特殊情况时效率更高。

分析与解答：

根据对此特例的观察，可将前缀函数修正为如下 next′：

$$\text{next}'[q]=\begin{cases}\text{next}'[\text{next}[q]] & p[\text{next}[q]+1]=p[q+1],\text{next}[q]>-1\\ \text{next}[q] & \text{其他}\end{cases}$$

相应的计算模式串的前缀函数的算法可修改如下。

```
1  private void mbuild(String p)
2  {//改进的前缀函数
3     int m=p.length();
4     int []f=new int[m];
5     build(p);
6     for(int i=0;i<m;i++) f[i]=next[i];
7     next[0]=-1;
8     for(int i=1;i<=m-1;i++){
9        int j=f[i];
10        if(j<0 || p.charAt(j)!=p.charAt(i)) next[i]=j;
11        else next[i]=next[j];
12     }
13  }
```

将主教材中的 build 换作 mbuild 就可用修正后的前缀函数来搜索子串，从而得到一个改进的 KMP 算法。

习题 9-4　确定所有匹配位置的 KMP 算法

修改 KMP 算法，使其能找到模式串 p 在主串 t 中的所有匹配位置。

分析与解答：

找到一个匹配位置后，可以利用前缀函数 next 的性质，继续比较 $t[i]$ 与 $p[\text{next}[j]+1]$。修改后的算法如下：

```
1  public void KMP_Matcher(String t)
2  {// KMP 算法
3     int m=p.length();
4     int n=t.length();
5     int j=-1;
6     for (int i=0;i<n;i++){
7        while(j>-1&& p.charAt(j+1)!=t.charAt(i)) j=next[j];
8        if(p.charAt(j+1)==t.charAt(i)) j++;
9        if(j==m-1){
10          System.out.println(i-m+1);
11          j=next[j];
12       }
13    }
14  }
```

习题 9-5　特殊情况下简单子串搜索算法的改进

假设模式串 p 中所有的字符均不相同。说明如何修改简单子串搜索算法，使其计算时间为 $O(n)$，其中 n 为主串 t 的长度。

分析与解答:

显而易见,如果 p 在 t 中多次出现,这些子串均不重叠。因此,当 $t[i+j]\neq p[j]$时,可以从 $t[i+j+1]$开始与 p 进行比较。这相当于 i 不再回溯。因为主串 t 中每个字符最多只比较 2 次,故总比较次数小于 $2n$,即计算时间为 $O(n)$。

修改后的算法如下:

```
1   public int NaiveSearch(String t,String p)
2   {//特殊情况下简单子串搜索算法
3     int m = p.length();
4     int n = t.length();
5     int i=0;
6     while(i<=n-m){
7       int j=0;
8       while(j<m && t.charAt(i+j)==p.charAt(j))j++;
9       if(j==m) return i;
10       i=i+j+1;
11    }
12    return -1;
13  }
```

习题 9-6　简单子串搜索算法的平均性能

设主串 t 和模式串 p 分别是从 $d(d\geqslant 2)$元字符集 $\sum_{d}=\{0,1,\cdots,d-1\}$ 中随机字符组成的长度为 n 和 m 的字符串。试证明简单子串搜索算法所做比较次数的期望值为

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}}\leqslant 2(n-m+1)$$

由此可见,对于随机选取的字符串,简单子串搜索算法还是十分有效的。

分析与解答:

在简单子串搜索算法中,对每个不同的 i(其中 $0\leqslant i\leqslant n-m$)有如下结论。$p[j]$需要与 $t[i+j]$进行比较的概率是 $p[0..j-1]=t[i..i+j-1]$的概率,即 $1/d^j$。也就是说,对每个 i,算法的期望比较次数均为 $\sum_{j=0}^{m-1}\frac{1}{d^j}$。因此,算法的总期望比较次数为$(n-m+1)\sum_{j=0}^{m-1}\frac{1}{d^j}$。由此可知

$$(n-m+1)\sum_{j=0}^{m-1}\frac{1}{d^j}=(n-m+1)\frac{d^{-m}-1}{d^{-1}-1}\leqslant(n-m+1)\frac{1}{1-d^{-1}}$$

$$\leqslant(n-m+1)\frac{1}{1-\frac{1}{2}}=2(n-m+1)$$

习题 9-7　带间隙字符的模式串搜索

假设允许模式串 p 中可以出现能与任意字符串(包括长度为 0 的空串)匹配的间隙字符◇。例如,模式串 ab◇ba◇c 可在主串 cabccbacbacab 产生如图 9-3 所示的匹配。

间隙字符◇可在模式串中出现任意多次,但不允许在主串中出现。试设计一个多项式

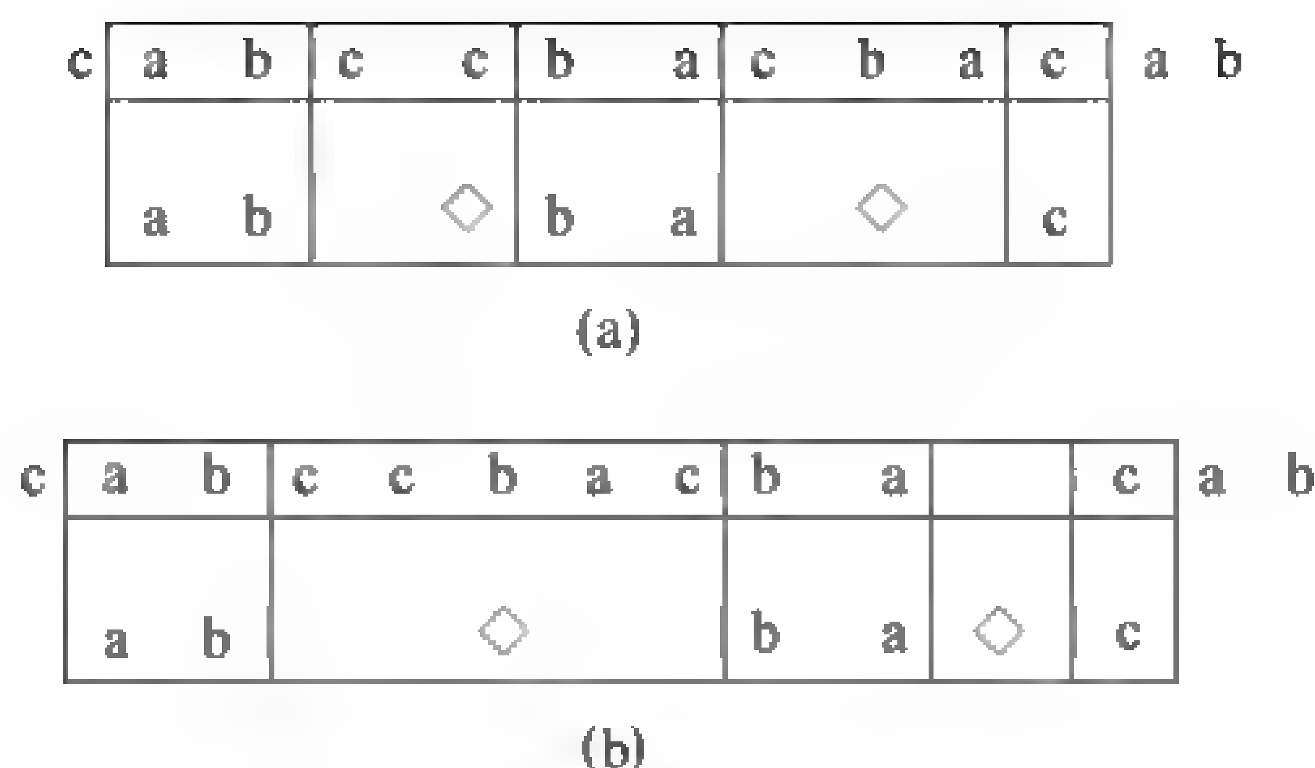


图 9-3　带间隙字符的模式串

时间算法，确定在主串中能否找到与模式串 p 匹配的子串，并分析算法的计算时间复杂性。

分析与解答：

由于◇可与主串中任意多个字符匹配，因此只要考查 p 中被◇分隔开的各个子串能否按序在主串 t 中找到匹配即可。这就转化成有限个 p 中的非◇字符子串在主串中的匹配问题。

假设 p 分解成 a_1◇a_2◇…◇a_k。首先找到 a_1 在主串中的最左端，接着从其右端开始找 a_2 的最左端，……，直到找到 a_k。如果用简单子串搜索算法实现这个过程，则需要 $O(nm)$ 计算时间。其中，n 是主串 t 的长度，m 是模式串 p 的长度。如果用 KMP 算法实现，则只需 $O(n+m)$ 计算时间。

习题 9-8　串接的前缀函数

设模式串 p 和主串 t 的串接为 pt。试说明如何利用 pt 的前缀函数来计算模式串 p 在主串 t 中出现的位置。

分析与解答：

如果 p 在 t 中出现，则存在 $k \geqslant 0$，使得 $t[k..k+m-1]=p[0..m-1]$。其中，m 为串 p 的长度。这说明 p 是 $pt[0..k+2m-1]$ 的真前缀，又是真后缀。因此，只须查找 pt 的前缀函数 next 的值。若存在 $i \geqslant 2m-1$ 且 $\text{next}[i] \geqslant m-1$，则 p 在位置 $i-\text{next}[i]$ 出现。也就是在 t 中位置 $i-\text{next}[i]-m$ 出现。

```
1  public int conc (String t,String p)
2  {//串接的前缀函数
3     int m=p.length();
4     int n=t.length();
5     next=new int[m+n];
6     String pt=p+t;
7     build(pt);
8     for(int i=0;i<n+m;i++)
9       if(i>=2*m-1 && next[i]>=m-1) return i-next[i]-m;
10    return -1;
11 }
```

另一个类似的方法是用一个在 p 和 t 中均未出现的字符 c 连接 p 和 t 为 pct。pct 的前缀函数 next 的值最大为 $m-1$。在任何一处出现 $\text{next}[i]=m-1$ 的位置 i，就是 p 出现的右

端位置。

```
1   public int conc(String t,String p)
2   {//串接的前缀函数
3     int m=p.length();
4     int n=t.length();
5     next=new int[m+n+1];
6     String pt=p+"*"+t;
7     build(pt);
8     for (int i=0;i<n+m+1;i++)
9       if(next[i]==m-1) return i-next[i]-m-1;
10        return -1;
11  }
```

习题 9-9　串的循环旋转

试设计一个线性时间算法,确定一个串 t 是否为另一串 t'的循环旋转。例如,arc 与 car 互为循环旋转。

分析与解答:

容易证明,t 是 t'的循环旋转,当且仅当 t'是 tt 的子串。

用 KMP 算法可在线性时间内计算如下。

```
1   public static boolean cyclic(String t,String p)
2   {//串的循环旋转
3     String t2=t+t;
4     int n=t2.length();int m=p.length();
5     KMP kmp=new KMP(t2);
6     return (n/2==m && kmp.search(p)>-1);
7   }
```

习题 9-10　失败函数性质

在字符串集合 P 的 AC 自动机 T 中,状态结点 s 所表示的字符串是从根结点到 s 的路径上各边的字符依次连接组成的字符串 $\alpha(s)$。设 s 和 t 是 T 中的两个结点,且 $u=\alpha(s)$,$v=\alpha(t)$。试证明,$f(s)=t$ 当且仅当 v 是字符串 p_i(其中 $0\leqslant i<k$)的所有前缀中 u 的最长真后缀。

分析与解答:

对 u 的长度$|u|$用数学归纳法。当$|u|=1$时,s 是 T 中第一层结点。对于 T 中所有的第一层结点 s',均有 $f(s')=0$,因而 $f(s)=0$,即 $t=0$,$v=\epsilon$。因此,当$|u|=1$时结论成立。

设结论对所有长度小于 j 的字符串成立。

当$|u|=j$时,设 $u=a_1a_2\cdots a_j$,且 v 是字符串 p_i(其中 $0\leqslant i<k$)的所有前缀中 u 的最长真后缀。进一步设状态结点 r 所表示的字符串是 $a_1a_2\cdots a_{j-1}$,s 所表示的字符串是 u。

$r_1,r_2,\cdots,r_q$ 是满足如下条件的结点序列

$$\begin{cases} r_1 = f(r) & \\ g(r_i,a_j) = -1 & 1\leqslant i<q \\ r_{i+1} = f(r_i) & 1\leqslant i<q \\ g(r_q,a_j) = t & \end{cases}$$

此序列实际上就是算法 build_failure 计算失败函数时产生的结点序列。算法在 while 循环结束后取 $f(s)=t$。

设 t 所表示的字符串为 v,则 v 就是字符串 p_i(其中 $0\leqslant i<k$)的所有前缀中 u 的最长真后缀。

事实上,设 r_i 所表示的字符串为 v_i(其中 $1\leqslant i\leqslant q$)。由归纳假设知,v_1 是 $a_1a_2\cdots a_{j-1}$ 的最长真后缀,v_2 是 v_1 的最长真后缀,…,v_q 是 v_{q-1} 的最长真后缀。因此,v_q 是 $a_1a_2\cdots a_{j-1}$ 的最长真后缀,且 v_qa_j 是 P 中某个字符串的前缀。从而 v_qa_j 是字符串 p_i(其中 $0\leqslant i<k$)的所有前缀中 u 的最长真后缀。算法 build_failure 中取 $f(s)=g(r_q,a_j)=t$,由数学归纳法即知结论成立。

习题 9-11　输出函数性质

设 s 是字符串集合 P 的 AC 自动机中的状态结点,且 $u=\alpha(s)$。试证明,$v\in$ output(s) 当且仅当 $v\in P$ 且 v 是 u 的后缀。

分析与解答:

从算法 insert 容易看出,输入字符串 p_i 后,在相应的叶结点 s_i 处 output(s_i)$=p_i$。因此,在算法计算转向函数 g 结束时,结论成立。

在算法计算失败函数阶段,可以对结点层数采用数学归纳法证明结论也成立。

在根结点处已经证明了结论成立。假设结论对所有层数小于 d 的结点结论成立。

设 s 是 d 层结点,且它表示的字符串是 u。对任一 $v\in$ output(s),如果 v 是算法计算转向函数时加入 output(s),则 $v\in P$。如果 v 是算法计算失败函数时加入 output(s),则 $v\in$ output($f(s)$)。而由习题 9-9 的结论知,v 是 u 的后缀。

反之设 $v\in P$ 是 u 的后缀。由于 $v\in P$,因此存在状态结点 t,使得 t 所表示的字符串是 v。由算法 insert 知,$v\in$ output(t)。因此,当 $v=u$ 时,$t=s$,自然有 $v\in$ output(s)。当 v 是 u 的真后缀时,由归纳假设可知 $v\in$ output($f(s)$)。在算法 build_failure 的第 15 行,output($f(s)$)合并到 output(s)。因此,$v\in$ output(s)。由数学归纳法即知结论成立。

习题 9-12　后缀数组类

试设计一个后缀数组类。用倍前缀算法构造后缀数组,并支持以下运算:

(1) length();　　　　//返回后缀数组长度

(2) select(int i);　　//返回 sa[i]

(3) index(int i);　　//返回 rank[i]

(4) llcp(int i);　　　//返回 lcp[i]

分析与解答:

用倍前缀算法构造后缀数组的类描述如下。

```
1   class SuffixDP
2   {//倍前缀算法构造后缀数组
3     static int maxn=13003;
4     int m,n=0;
5     int [ ] a=new int[maxn];
6     int [ ] b=new int[maxn];
7     int [ ] cnt=new int[maxn];
```

```
8    int [ ] t=new int[maxn];
9    int [ ] sa=new int[maxn];
10   int [ ] rank=new int[maxn];
11   int [ ] lcp=new int[maxn];
12
13   public SuffixDP(String txt)
14   {
15     m=128;n=txt.length();
16     for(int i=0;i<n;i++)t[i]=txt.charAt(i);
17     doubling(t);
18     kasai(t,n);
19   }
20   public int length(){return n;}
21   public int select(int i){return sa[i];}
22   public int index(int i){return rank[i];}
23   public int llcp(int i){return lcp[i];}
24 }
```

习题 9-13　最长公共扩展查询

试说明如何对最长公共前缀数组 lcp 做适当预处理，使得最长公共扩展查询在最坏情况下需要 $O(1)$时间。

分析与解答：

首先注意到，如果事先计算好最长公共前缀数组 lcp 所有查询 rmq(i,j)，并存入一个二维数组 dp，就可以做到 $O(1)$时间响应。例如：

```
1  class RMQ
2  {//区间最小查询
3    int n=0;
4    int []a;int [][]dp;
5
6    public RMQ(int n,int []a)
7    {
8      this.n=n;
9      this.a=a;
10     dp=new int[n][n];
11     build();
12   }
13
14   private void build()
15   {
16     for(int i=0;i<n;i++)dp[i][i]=i;
17     for(int i=0;i<n;i++)
18       for(int j=i+1;j<n;j++)
19         if(a[dp[i][j-1]]>a[j])dp[i][j]=j;
20     else dp[i][j]=dp[i][j-1];
```

```
21    }
22
23    public int query(int l,int r)
24    {
25        return a[dp[l][r]];
26    }
27  }
```

其中,build 用动态规划算法将所有查询 rmq(i,j)保存到数组 dp 中。查询时,直接返回查询值。预处理时间为 $O(n^2)$,需要空间也是 $O(n^2)$。

用稀疏表(Sparse Table)算法可以将预处理时间和空间降为 $O(n\log n)$。其基本思想是,对(i,j)(其中 $0\leqslant i<n-1$,$0\leqslant j<\log(n)$),将区间[i,2^j]的最小值保存在 dp[j][i]中。数组 dp 需要 $O(n\log n)$空间,用动态规划算法可以在 $O(n\log n)$时间计算数组 dp。

$$\mathrm{dp}[j][i]=\begin{cases}\mathrm{dp}[j-1][i] & \mathrm{lcp}[\mathrm{dp}[j-1][i]\leqslant \mathrm{dp}[j-1][i+2^{j-1}-1]\\ \mathrm{dp}[j-1][i+2^{j-1}-1] & \text{其他}\end{cases}$$

在响应查询 rmq(i,j)时,选择覆盖查询区间[i,j]的两个已经计算和最小值的区间如下。

设 $k=[\log(j-i)]$,则[i,$i+2^k-1$]和[$j-2^k+1$,j]覆盖查询区间[i,j]。因此,min{dp[k][i],dp[k][$j-2^k+1$]}就是 rmq(i,j)的值。

稀疏表算法实现如下:

```
1  class RMQ
2  {//区间最小查询稀疏表算法
3    int n=0;
4    int []a;int [][]dp;
5
6    public RMQ(int n,int []a)
7    {
8        this.n=n;
9        this.a=a;
10       dp=new int[n][20];
11       prep();
12   }
13
14   private void prep()
15   {
16       int k=(int)Math.floor(Math.log(n)/Math.log(2.0));
17       for(int i=0;i<n;i++)dp[i][0]=a[i];
18       for(int i=1;i<=k;i++)
19         for(int j=0;j+(1<<i)-1<n;j++)
20           dp[j][i]=Math.min(dp[j][i-1],dp[j+(1<<(i-1))][i-1]);
21   }
22
23   public int rmq(int l,int r)
```

```
24    {
25      int k=(int)Math.floor(Math.log(r-l+1)/Math.log(2.0));
26      return Math.min(dp[l][k],dp[r-(1<<k)+1][k]);
27    }
28  }
```

用区间块分割算法可以将预处理时间和空间降为$O(n)$，但是查询响应时间增加为$O(\sqrt{n})$。其基本思想是将数组划分为$\sqrt{n}$块，每块中有$\sqrt{n}$个元素，事先计算好每块的最小值，查询时先按块查询，然后再块内查询。

```
1   class RMQsp
2   {//区间最小查询区间块分割算法
3     int n=0;
4     int []a;int []sp;
5     public RMQsp(int n,int []a)
6     {
7       this.n=n;
8       this.a=a;
9       sp=new int[n];
10       build();
11    }
12    private void build()
13    {
14       int size_m =0;
15       for(int i=0;i<n;){
16         int minindex = i;
17         for(int j=0;j<(int)Math.sqrt(n) && i<n;j++){
18           if(a[i]<a[minindex]) minindex=i;
19           i++;
20         }
21         sp[size_m++]=minindex;
22       }
23    }
24    public int query(int l,int r)
25    {
26       if(l==r)return a[l];
27       int b=l/(int)Math.sqrt(n),e=r/(int)Math.sqrt(n);
28       int ans=a[l];
29       for(int i=l;i<(b+1)*Math.sqrt(n);i++)if(a[i]<ans)ans=a[i];
30       for(int i=b+1;i<e;i++)if(a[sp[i]]<ans)ans=a[sp[i]];
31       for(int i=e*(int)Math.sqrt(n);i<=r;i++)if(a[i]<ans)ans=a[i];
32       return ans;
33    }
34  }
```

用序列树来表示查询区间可以将预处理时间和空间降为$O(n)$，但是查询响应时间为

$O(\log n)$。

```
1  class RMQseg
2  {//区间最小查询序列树算法
3    int n=0;
4    int []a;int []seg;
5    public RMQseg(int n,int []a)
6    {
7      this.n=n;
8      this.a=a;
9      System.out.println("n="+n);
10      seg=new int[2*n+5];
11      build(0,0,n-1);
12    }
13    private void build(int node,int start,int end)
14    {
15      System.out.println("node="+node+" "+2*node+2);
16      if(start==end) seg[node]=start;
17      else{
18        build(2*node+1,start,(start+end)/2);
19        build(2*node+2,(start+end)/2+1,end);
20        if(a[seg[2*node+1]]<a[seg[2*node+2]]) seg[node]=seg[2*node+1];
21        else seg[node]=seg[2*node+2];
22      }
23    }
24    private int search(int node,int start,int end,int s,int e)
25    {
26      if(s<=start && e>=end) return seg[node];
27      else if(s>end || e<start) return -1;
28      int q1=search(2*node+1,start,(start+end)/2,s,e);
29      int q2=search(2*node+2,(start+end)/2+1,end,s,e);
30      if(q1==-1) return q2;
31      else if(q2==-1) return q1;
32      if(a[q1]<a[q2]) return q1;
33      return q2;
34    }
35    public int query(int s,int e)
36    {
37      int ret=search(0,0,n-1,s,e);
38      if(ret>=0) return a[search(0,0,n-1,s,e)];
39      else return Integer.MAX_VALUE;
40    }
41  }
```

在计算最长公共扩展时，先计算后缀数组和最长公共前缀数组 lcp，然后建立 lcp 的 RMQ 类，这样就可以在 $O(1)$时间内响应最长公共扩展查询。

习题 9-14　最长公共扩展性质

设字符串 t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。对于非负整数 $0\leqslant l\leqslant r$,t 的后缀 S_l 和 S_r 的最长前缀的长度为 $\text{lce}(l,r)$。设 $x=\text{sa}^{-1}[l]$,$z=\text{sa}^{-1}[r]$,则 $\text{sa}[x]=l$,$\text{sa}[z]=r$。不失一般性,可设 $x<z$。试证明 $\text{lce}(l,r)$ 具有如下性质:

$$\text{lce}(l,r)=\min_{x\leqslant y<z}\{\text{lce}(\text{sa}[y],\text{sa}[y+1])\}=\min_{x\leqslant y<z}\{\text{lcp}[y]\} \tag{9.1}$$

分析与解答:

首先注意到,对于任意的 $0\leqslant x<y<z$,有

$$\text{lce}(\text{sa}[x],\text{sa}[z])=\min\{\text{lce}(\text{sa}[x],\text{sa}[y]),\text{lce}(\text{sa}[y],\text{sa}[z])\}=\delta \tag{9.2}$$

事实上,

(1) 按照 δ 的定义有

$$\begin{cases}t[\text{sa}[x]..\text{sa}[x]+\delta-1]=t[\text{sa}[y]..\text{sa}[y]+\delta-1]\\ t[\text{sa}[y]..\text{sa}[y]+\delta-1]=t[\text{sa}[z]..\text{sa}[z]+\delta-1]\end{cases} \tag{9.3}$$

从而 $t[\text{sa}[x]..\text{sa}[x]+\delta-1]=t[\text{sa}[z]..\text{sa}[z]+\delta-1]$,即

$$\text{lce}(\text{sa}[x],\text{sa}[z])\geqslant\delta \tag{9.4}$$

(2) 由 $0\leqslant x<y<z$ 可知,$S_{\text{sa}[x]}<S_{\text{sa}[y]}<S_{\text{sa}[z]}$。因此有

$$t[\text{sa}[x]+\delta]\leqslant t[\text{sa}[y]+\delta]\leqslant t[\text{sa}[z]+\delta]$$

如果 $t[\text{sa}[x]+\delta]=[\text{sa}[z]+\delta]$,则必有 $t[\text{sa}[x]+\delta]=[\text{sa}[y]+\delta]=[\text{sa}[z]+\delta]$。

由此可得,$\text{lce}(\text{sa}[x],\text{sa}[y])>\delta$ 且 $\text{lce}(\text{sa}[y],\text{sa}[z])>\delta$,这与 δ 的定义矛盾。由此可见,$t[\text{sa}[x]+\delta]<[\text{sa}[z]+\delta]$。也就是说,

$$\text{lce}(\text{sa}[x],\text{sa}[z])\leqslant\delta \tag{9.5}$$

结合式(9.4)和式(9.5)即知,$\text{lce}(\text{sa}[x],\text{sa}[z])=\delta$。

据此对 $z-x$ 用数学归纳法就可以证明式(9.1)。事实上,当 $z-x=1$ 时,式(9.1)显然成立。假设当 $z-x\leqslant m$ 时式(9.1)成立。当 $z-x=m+1$ 时,任取 $x<p<z$,则有 $z-p\leqslant m$ 且 $p-x\leqslant m$。由式(9.2)和归纳假设即知,

$$\begin{aligned}\text{lce}(\text{sa}[x],\text{sa}[z])&=\min\{\min_{x\leqslant y<p}\{\text{lce}(\text{sa}[y],\text{sa}[y+1])\},\min_{p\leqslant y<z}\{\text{lce}(\text{sa}[y],\text{sa}[y+1])\}\}\\&=\min_{x\leqslant y<p}\{\text{lce}(\text{sa}[y],\text{sa}[y+1])\}\end{aligned}$$

由数学归纳法即知式(9.1)成立。

习题 9-15　后缀数组性质

设字符串 t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。数组 h 定义为 $h[i]=\text{lcp}[\text{sa}^{-1}[i]]$,$0\leqslant i\leqslant n-2$。试证明,如果 $h[i]>1$,则

$$h[i+1]\geqslant h[i]-1 \tag{9.6}$$

分析与解答:

设 $p=\text{rank}[i]$,$q=\text{rank}[i+1]$,$j=\text{sa}[p+1]$,$k=\text{sa}[q+1]$。

(1) 有以下算式成立:

$$h[i+1]\geqslant h[i]-1\Leftrightarrow\text{lce}(i+1,k)\geqslant\text{lce}(i,j)-1 \tag{9.7}$$

事实上,由 h 和 lcp 的定义即知,

$$\begin{cases}h[i]=\text{lcp}[p]=\text{lce}(\text{sa}[p],\quad\text{sa}[p+1])=\text{lce}(i,j)\\ h[i+1]=\text{lcp}[q]=\text{lce}(\text{sa}[q],\quad\text{sa}[q+1])=\text{lce}(i+1,k)\end{cases} \tag{9.8}$$

(2) 从式(9.1)可以看出，

$$x < y \leqslant z \rightarrow \text{lce}(\text{sa}[x],\text{sa}[z]) \leqslant \text{lce}(\text{sa}[y],\text{sa}[z]) \tag{9.9}$$

(3) 如果 $\text{lce}(i,j)>1$，则有

$$\text{lce}(i+1,k) \geqslant \text{lce}(i+1,j+1) \tag{9.10}$$

事实上，由 $p=\text{rank}[i]<p+1=\text{rank}[j]$ 可知 $S_i<S_j$。又由于 $\text{lce}(i,j)>1$ 知 $t[i]=t[j]$。因此，可得 $S_{i+1}<S_{j+1}$，即 $\text{rank}[i+1]<\text{rank}[j+1]$。这等价于 $\text{rank}[i+1]+1\leqslant \text{rank}[j+1]$。由于 $\text{rank}[i+1]+1=\text{rank}[k]$，所以有

$$\text{rank}[i+1] < \text{rank}[k] \leqslant \text{rank}[j+1] \tag{9.11}$$

根据式(9.9)的结论就有 $\text{lce}(i+1,k)\geqslant \text{lce}(i+1,j+1)$。

(4) 根据式(9.8)的结论可知，

$$\begin{cases} h[i] = \text{lce}(i,j) > 1 \\ h[i+1] = \text{lce}(i+1,k) \end{cases} \tag{9.12}$$

根据式(9.10)的结论有 $\text{lce}(i+1,k)\geqslant \text{lce}(i+1,j+1)=\text{lce}(i,j)-1$。换句话说，$h[i+1]\geqslant h[i]-1$。

习题 9-16　后缀数组搜索

设字符串 t 和 p 的长度分别为 m 和 n。t 的后缀数组为 sa。请说明如何利用 t 的后缀数组搜索给定字符串 p 在 t 中出现的所有位置。要求算法在最坏情况下的时间复杂性为 $O(m\log n)$。

分析与解答：

由于 t 的后缀数组 sa 是排好序的，字符串 p 在 t 中出现的所有位置在后缀数组 sa 中是连续排列的。因此，可以对后缀数组 sa 用二分搜索算法找到这个连续排列的开始位置和结束位置。首先用二分搜索算法找到 sa 的最小位置 i，使得 p 是 $\text{sa}[i]$的前缀。如果找不到最小位置 i，则说明 t 不含字符串 p；否则从最小位置 i 开始用二分搜索算法找到 sa 的最大位置 j，使得 p 是 $\text{sa}[j]$的前缀。这样就找到了 p 在 t 中出现的所有位置 $\text{sa}[i..j]$。

具体算法描述如下：

```
1   private int lower(String p)
2   {//后缀数组最小位置搜索
3     int lft=0,len=text.length();
4     while(len>0){
5       int half=len>>1,mid=lft+half;
6       int res=cmp(p,mid);
7       if(res>0){lft=mid+1;len=len-half-1;}
8       else len=half;
9     }
10    return lft;
11  }
```

```
1   private int upper(String p,int lft)
2   {//后缀数组最大位置搜索
3     int len=text.length()-lft-1;
4     while(len>0){
```

```
5        int half=len>>1,mid=lft+half;
6        int res=cmp(p,mid);
7        if(res<0)len=half;
8        else{lft=mid+1;len=len-half-1;}
9      }
10     return lft;
11  }
```

算法 lower 和 upper 分别用于寻找 sa 的最小位置 i 和最大位置 j。它们与第 2 章中介绍的二分搜索算法如出一辙，其中的比较函数 cmp 用于比较 $S_{sa[mid]}$ 和 p。

```
1  private int cmp(String p,int j)
2  {//比较函数
3    return p.compareTo(text.substring(sa[j],Math.min(n,sa[j]+m)));
4  }
```

结合 lower 和 upper 就可以找到 p 在 t 中出现的所有位置。

```
1  public int search(String p)
2  {//后缀数组搜索
3    m=p.length();
4    int l=lower(p);
5    if(p.compareTo(text.substring(sa[l],Math.min(n,sa[l]+m)))!=0) return -1;
6    int r=upper(p,l);
7    return r-l;
8  }
```

在算法的第 4 行判断 t 是否含字符串 p。由于在最坏情况下比较函数 cmp 需要 $O(m)$ 时间，因此整个算法需要的计算时间是 $O(m\log n)$。

习题 9-17　后缀数组快速搜索

设字符串 t 和 p 的长度分别为 m 和 n。t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。试说明如何利用 t 的后缀数组和最长公共前缀数组，搜索给定字符串 p 在 t 中出现的所有位置。要求算法在最坏情况下的时间复杂性为 $O(m+\log n)$。

分析与解答：

在一般情况下，设要搜索的后缀数组区间是 sa$[L,R]$。开始时 $L=0,R=n-1$。在搜索过程中始终有

$$S_{sa[L]} \leqslant p \leqslant S_{sa[R]} \tag{9.13}$$

仍然用习题 9-16 中的二分搜索算法。关键运算是比较 p 和 $S_{sa[M]}$ 的大小，即计算 $\delta(p,S_{sa[M]})$ 的值。其中 $M=L+R$ 是搜索区间的中点。

$$\delta(p,S_{sa[M]})=\begin{cases}-1 & p>S_{sa[M]}\\ 0 & p=S_{sa[M]}\\ 1 & p<S_{sa[M]}\end{cases} \tag{9.14}$$

在搜索过程中设

$$
\begin{cases}
l = \mathrm{lcp}(p, S_{\mathrm{sa}[L]}) \\
r = \mathrm{lcp}(p, S_{\mathrm{sa}[R]}) \\
\mathrm{mid} = \mathrm{lcp}(p, S_{\mathrm{sa}[M]}) \\
\alpha = \mathrm{lcp}(S_{\mathrm{sa}[L]}, S_{\mathrm{sa}[M]}) \\
\beta = \mathrm{lcp}(S_{\mathrm{sa}[M]}, S_{\mathrm{sa}[R]})
\end{cases}
\tag{9.15}
$$

其中,$\mathrm{lcp}(p,q)$是字符串 p 和 q 的最长公共前缀的长度。

由于后缀数组 sa 是按照 t 的所有后缀的字典序排好序的,$S_{\mathrm{sa}[M]}$ 和 p 的前 $\min(l,r)$ 个字符相同。

当 $l=r$ 时,只要从 $S_{\mathrm{sa}[M]}$ 和 p 的第 $l+1$ 个字符开始比较。

当 $l>r$ 时,有以下 3 种情形:

(1) 当 $l<\alpha$ 时,$p[l]>t[\mathrm{sa}[L]+l]=t[\mathrm{sa}[M]+l]$。因此有

$$
\begin{cases}
p > S_{\mathrm{sa}[M]} \\
\mathrm{mid} = l
\end{cases}
$$

搜索区间改变为 $\mathrm{sa}[M,R]$,无须比较字符。

(2) 当 $l>\alpha$ 时,$p[\alpha]=t[\mathrm{sa}[L]+\alpha]<t[\mathrm{sa}[M]+\alpha]$。因此有

$$
\begin{cases}
p < S_{\mathrm{sa}[M]} \\
\mathrm{mid} = \alpha
\end{cases}
$$

搜索区间改变为 $\mathrm{sa}[L,M]$,无须比较字符。

(3) 当 $l=\alpha$ 时,$S_{\mathrm{sa}[M]}$ 和 p 的前 l 个字符相同,所以只要从 $S_{\mathrm{sa}[M]}$ 和 p 的第 $l+1$ 个字符开始比较即可。

当 $l<r$ 时,有以下 3 种情形:

(1) 当 $r<\beta$ 时,有 $p[r]<t[\mathrm{sa}[R]+r]=t[\mathrm{sa}[M]+r]$。因此有

$$
\begin{cases}
p < S_{\mathrm{sa}[M]} \\
\mathrm{mid} = r
\end{cases}
$$

搜索区间改变为 $\mathrm{sa}[L,M]$,无须比较字符。

(2) 当 $r>\beta$ 时,有 $p[\beta]=t[\mathrm{sa}[R]+\beta]>t[\mathrm{sa}[M]+\beta]$。因此有

$$
\begin{cases}
p > S_{\mathrm{sa}[M]} \\
\mathrm{mid} = \beta
\end{cases}
$$

搜索区间改变为 $\mathrm{sa}[M,R]$,无须比较字符。

(3) 当 $r=\beta$ 时,$S_{\mathrm{sa}[M]}$ 和 p 的前 r 个字符相同,所以只要从 $S_{\mathrm{sa}[M]}$ 和 p 的第 $r+1$ 个字符开始比较即可。

综合以上分析可知,$\delta(p,S_{\mathrm{sa}[M]})$可以根据参数 l,r,α,β,M 来计算。

设

$$
\eta = \delta(p[\max(l,r),m-1],t[\mathrm{sa}[M]+\max(l,r),n-1]) \tag{9.16}
$$

则

$$
\delta(p,S_{\mathrm{sa}[M]}) = \begin{cases}
-1 & r<l<\alpha \lor r>\max(l,\beta) \\
1 & l<r<\beta \lor l>\max(r,\alpha) \\
\eta & l=r \lor l<r=\beta \lor r<l=\alpha
\end{cases}
\tag{9.17}
$$

将习题 9-16 中找下界和上界的算法 lower 和 upper 的比较函数 cmp 换成式(9.17)所

表示的比较函数如下：

```
1  private int cmp(String p,int lft,int rht,int mid)
2  {//改进的比较函数
3    int al=lce(lft,mid),be=lce(mid,rht);
4    int l=lr[0],r=lr[1];
5    if(r<l && l<al || r>Math.max(l,be))return -1;
6    else if(l<r && r<be || l>Math.max(r,al))return 1;
7    else{
8    int k=Math.max(l,r);
9    int id=l==k?0:1;
10       int ret=cp(p,id,sa[mid]);
11       if (ret>0) lr[0]=k;
12       else lr[1]=k;
13       return ret;
14     }
15 }
```

其中，变量 lft，rht，mid 分别对应于搜索区间的左端点、右端点和中点。变量 l，r，al，be 分别对应于式(9.15)中的 l，r，α，β。在算法的第 3 行用函数 lce 来计算 α 和 β。在第 8 行按照式(9.16)用 cp 来计算 η。

```
1  private int lce(int l,int r)
2  {// 计算α和β值
3    if(l==r)return(n-sa[l]);
4    if(r>n-1)return 0;
5    return rq.rmq(l,r-1);
6  }
```

在计算 lce 时，先要用数组 lcp 做一些预处理。建立对数组 lcp 做区域最小查询的类 rq。rq.query(l,r)可以在 $O(1)$时间内计算 lcp[l,r]中最小值，而这个最小值就是 $S_{sa[l]}$ 和 $S_{sa[r]}$ 的最长公共前缀。

```
1  private int cp(String p,int id,int j)
2    {// 计算η值
3      int k=lr[id];
4      while(k<m && k+j<n && text.charAt(k+j)==p.charAt(k))k++;
5      lr[id]=k;
6      if(k==m) return (n>k+j? 1:0);
7      else if(k+j==n) return -1;
8      else return (int)(text.charAt(k+j)-p.charAt(k));
9  }
```

cp 在计算 η 时，从第 k 个字符开始比较字符串 p 和 $S_{sa[j]}$。计算结束后返回 k 的值。

要用改进的比较函数 cmp 来计算，还需要对算法 lower 和 upper 做一些改变，改进后的算法如下：

```
1  private int lower(String p)
2  {//后缀数组最小位置搜索
3    int lft=0,rht=n-1,len=n;
4    lr[0]=0;lr[1]=0;
5    if(cp(p,0,sa[0])>0 || cp(p,1,sa[n-1])<0)return 0;
6    while(len>0){
7      int half=len>>1,mid=lft+half;
8      int res=cmp(p,lft==0? lft:lft-1,len==n? rht:rht+1,mid);
9      if(res==0)return mid;
10      if(res<0){lft=mid+1;len=len-half-1;}
11      else{len=half;rht=lft+len-1;}
12    }
13    return lft;
14 }
```

首先在第4行计算 $S_{sa[0]}$ 和 $S_{sa[n-1]}$ 与 p 的最长前缀长度，并判断式(9.13)的条件是否满足。这个条件是整个算法中必须满足的不变性条件。只有满足这个条件，才能保证公式(9.17)的正确性。在算法的第8行，没有直接用lft和rht的值来计算cmp的值。这是因为算法在改变搜索区间时，并不是用mid来替换lft或rht，而是用mid+1来替换lft，或者用mid−1来替换rht。这样就可能破坏了不变性条件式(9.13)。因而，在计算cmp的值时，lft需要向左退一步，rht需要向右退一步。当 $t[n-m,n-1]=p$ 时，出现cmp值为0。继续搜索就可能破坏了不变性条件式(9.13)。不过此时已经找到下界，可以终止搜索，所以在第9行直接返回其值。算法的其他部分不变。

```
1  private int upper(String p,int fst)
2  {//后缀数组最大位置搜索
3    int lft=fst,rht=n-1,len=n-lft-1;
4    lr[0]=0;lr[1]=0;
5    while(len>0){
6      int half=len>>1,mid=lft+half;
7      int lft1=lft==fst? lft:lft-1,rht1=len==n? rht:rht+1;
8      int lm=lce(lft1,mid);
9      int res=(cmp(p,lft1,rht1,mid)>0 && (lm<m))? 1:0;
10      if(res>0) {len=half;rht=lft+len-1;}
11      else {lft=mid+1;len=len-half-1;}
12    }
13    return lft;
14 }
```

算法upper的主要改变在第7～8行。由于要找的是后缀数组sa的上界，即返回值lft是使得sa[lft]>p的最小元素。因此，$S_{sa[lft]}$ 与 p 的最长公共前缀的长度小于 m。而此条件在cmp返回的对于0的值中还不能体现。但是，可以通过第7行中计算的值来判断 $S_{sa[mid]}$ 与 p 的最长公共前缀的长度是否小于 m。因此，在第8行中还要加此条件。

改进后的算法在 lower 的第4行最多做了 $2m$ 次比较。将改进的比较函数 cmp 所做的比较分为两类，即相等比较和不相等比较。算法的每一次迭代最多有1次不相等比较，而迭代次数不超过 $\log n$ 次。所以，不相等比较次数不超过 $\log n$。对于相等比较，注意到每次比较都是从 p 的第 $k=\max\{l,r\}+1$ 个字符开始，而且每次相等比较都使下一轮迭代的 $\max\{l,r\}$ 值增加。由此可见，p 的每个字符最多只参加1次相等比较。也就是说，相等比较次数不超过 m。由此可知，改进的比较函数 cmp 所做的比较次数不超过 $m+\log n$。算法 lower 和 upper 循环体内其他运算均需 $O(1)$ 时间。因此，算法需要的总时间是 $O(m+\log n)$。

算法实现题 9-1　安全基因序列问题

★ 问题描述

基因序列是用字符串表示的携带基因信息的 DNA 分子的一级结构。基因序列的字符集是 $\Sigma=\{A,C,G,T\}$。其中字符分别代表组成 DNA 的4种核苷酸：腺嘌呤、胞嘧啶、鸟嘌呤和胸腺嘧啶。许多疾病往往是由基因突变引起的。这种基因突变是从一个正常的基因序列，通过几代人的遗传而产生的。对于基因片段的分析，有助于了解基因突变导致的遗传疾病。例如，如果一个基因序列中含有基因片段 ATG，则可能含有某种遗传疾病。生物科学家们已经发现许多这类基因片段。对于已知的、不安全的基因片段集合 P，如果一个基因序列中含有 P 中基因片段，则称该基因序列为不安全的基因序列；否则，称该基因序列为安全的基因序列。

★ 算法设计

对于给定的不安全的基因片段集合 P 以及一个正整数 n，计算长度为 n 的安全的基因序列个数。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第1行有两个正整数 n（其中 $1\leqslant n\leqslant 2\,000\,000\,000$）和 m（其中 $0\leqslant m\leqslant 10$），n 是基因序列长度，m 是不安全的基因片段个数。接下来的 m 行中，每行是一个长度不超过10的不安全的基因片段。

★ 结果输出

将计算出的长度为 n 的、安全的基因序列个数 mod 10 000，输出到文件 output.txt 中。

输入文件示例	输出文件示例
3 4	36
AT	
AC	
AG	
AA	

分析与解答：

首先建立不安全的基因片段集合 P 的 AC 自动机 T。如果一个基因序列 t 含有 P 中不安全的基因片段，则用 AC 自动机 T 做关于基因序列 t 的多子串搜索就可以找出它所包含的所有不安全的基因片段。反之，如果搜索不到不安全的基因片段，则基因序列 t 是安全的基因序列。基于这个基本思想，在 T 的基础上构造一个有向无环图 T' 如下。在 T 中删去

所有非空 output，即 cnt>0 的结点，得到 T'。由此可知，从根结点到 T' 中任一结点的路径组成的字符串都是安全的基因序列。设 T' 有 m 个结点，且 T' 的邻接矩阵为 $a[0..3][0..m-1]$。邻接矩阵为 $\boldsymbol{a}$ 的第 1 列中 4 个元素 $a[0..3][0]$ 分别表示从根结点出发经过边 A，C，G，T 的且长度为 1 的字符串个数。$\boldsymbol{a}^2$ 的第 1 列中 4 个元素 $a^2[0..3][0]$ 分别表示从根结点出发经过边 A，C，G，T 的且长度为 2 的字符串个数。以此类推，$\boldsymbol{a}^n$ 的第 1 列中 4 个元素 $a^n[0..3][0]$ 分别表示从根结点出发经过边 A，C，G，T 的且长度为 n 的字符串个数。由此可知，$\sum_{i=0}^{3} a^n[i][0]$ 就是长度为 n 的安全的基因序列个数。按此思路，解题需要经过以下 3 个步骤：

(1) 建立不安全的基因片段集合 P 的 AC 自动机 T。

(2) 根据 AC 自动机 T 建立 T'的邻接矩阵 $\boldsymbol{a}$。

(3) 计算 $\boldsymbol{a}^n$，并输出 $\sum_{i=0}^{3} a^n[i][0]$。

由于本题只要输出安全的基因序列个数，在不安全的基因片段集合 P 的 AC 自动机 T 中可以不必存储输出函数 output，因而可以用简化版的 AC 自动机 T。

根据 AC 自动机 T 建立 T'的邻接矩阵 $\boldsymbol{a}$ 如下：

```
1  public void buildadj(int [][]adj)
2  {//建立邻接矩阵
3    for(int i=0;i<size;i++)
4      for(int j=0;j<dsize;j++){
5        node tmp=nmap[i].go[j];
6        if(nmap[i].cnt==0 && tmp.cnt==0)adj[i][tmp.state]++;
7      }
8  }
```

用二分法计算矩阵幂的算法如下：

```
1  private int pow(int n)
2  {//计算矩阵幂
3    int [][]adj=new int[size][size];
4    int [][]pw=new int[size][size];
5    for(int i=0;i<size;i++)
6      for(int j=0;j<size;j++)pw[i][j]=i==j? 1:0;
7    buildadj(adj);
8    binpow(pw,adj,size,n);
9    int ans=0;
10    for(int i=0;i<size;i++)ans=(pw[0][i]+ans)%mod;
11    return ans;
12 }
```

```
1  private void binpow(int [][]t,int [][]a,int sz,int n)
2  {//用二分法计算矩阵幂
```

```
3     while(n>0){
4       if(n%2==1)mmult(t,a,sz);
5       mmult(a,a,sz);
6       n/=2;
7     }
8   }
```

上面的算法中,mmult 用于计算两个矩阵的乘积。

综合以上步骤,计算安全的基因序列个数 $\sum_{i=0}^{3} a^n[i][0]$ 的算法描述如下:

```
1  public static void comp(List<String> keywords,int n)
2  {//计算安全的基因序列个数
3     Ex91 ac=new Ex91(keywords);
4     System.out.println(ac.pow(n));
5  }
```

上面的算法中,keywords 是不安全的基因片段集合。

算法实现题 9-2　最长重复子串问题

★ 问题描述

最长重复子串问题在分子生物学和模式识别中有着广泛的应用。这个问题可以具体表述如下:给定 1 个长度为 n 的 DNA 序列 X,最长重复子串问题就是要找出在 X 中出现两次以上且长度最长的子串。例如,如果给定的 DNA 序列为 X=AGCATGCATGCAT,则子串 GCATGCAT 是 X 的一个最长重复子串,它在 X 的位置 1 和 5 处出现(第 1 个字符的位置为 0)。

★ 算法设计

设计一个算法,找出给定字符串 X 的最长重复子串。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中给出字符串 X。

★ 结果输出

将计算出的字符串 X 的最长重复子串输出到文件 output.txt 中。

文件的第 1 行是最长重复子串的长度。文件的第 2 行是最长重复子串。

输入文件示例	输出文件示例
AGCATGCATGCAT	8
	GCATGCAT

分析与解答:

此题是后缀数组的一个简单应用。先计算出 X 的后缀数组 sa 和最长公共前缀数组 lcp。

最长重复子串的长度实际上就是 X 的两个后缀的最长公共前缀的最大值。所以,最长公共前缀数组 lcp 中最大值就是 X 的最长重复子串的长度。如果在 lcp[i]取得最大值,则最长重复子串就是 X 从位置 sa[i]开始的长度为 lcp[i]的子串。

```
1   public void lrs(String s)
2   {//最长重复子串
3      int n=s.length(),len=0,j=-1;
4      SuffixDC3 suf=new SuffixDC3(s);
5      int []sa=suf.sa;
6      int []lcp=suf.lcp;
7      for(int i=0;i<n-1;i++)
8        if(lcp[i]>len){
9          len=lcp[i];j=i;
10       }
11     String t=s.substring(sa[j],Math.min(n,sa[j]+len));
12     System.out.println(len);
13     System.out.println(t);
14  }
```

计算 X 的后缀数组 sa 和最长公共前缀数组 lcp 需要 $O(n)$时间。计算最长公共前缀数组 lcp 中最大值显然只需 $O(n)$时间。

由此可知,上述最长重复子串算法需要 $O(n)$时间。

算法实现题 9-3　最长回文子串问题

★ 问题描述

如果一个字符串正读和反读相同,则称此字符串为回文。如果字符串 X 的一个子串 Y 是回文,则称子串 Y 是字符串 X 的一个回文子串。最长回文子串问题可以具体表述如下:给定 1 个长度为 n 的字符串 X,最长回文子串问题就是要找出 X 中长度最长的回文子串。例如,如果给定的字符串 X=bbacababa,则子串 bacab 是 X 的一个最长的回文子串,它的长度是 5。

★ 算法设计

设计一个算法,找出给定字符串 X 的最长回文子串。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中给出字符串 X。

★ 结果输出

将计算出的字符串 X 的最长回文子串输出到文件 output.txt 中。文件的第 1 行是最长回文子串的长度。文件的第 2 行是最长回文子串。

输入文件示例	输出文件示例
bbacababa	5
	bacab

分析与解答

当回文串的长度是一个奇数 $2k+1$ 时,第 $k+1$ 个字符就称为该回文串的中心;当回文串的长度是一个偶数 $2k$ 时,它的中心定义为第 k 和第 $k+1$ 个字符之间的位置。在这两种情形都定义回文串的半径为 k。注意到每个不同的最长回文子串的中心也不同。因此,最长回文子串的中心最多出现在 $2n-1$ 个不同位置。设给定字符串 s 的逆串为 $s^{\sim}$。构造一

个新字符串 $t=s\$s^{\sim}$,其中 \$ 是不在 s 中的特殊的字符。计算新字符串 t 的后缀数组 sa 和最长公共前缀数组 lcp,这样就把问题变成计算新字符串 t 的最长公共扩展问题。如果最长回文子串的长度是一个奇数,且其中心位置是 i,则其长度是 $s[i+1,n-1]$ 与 $s^{\sim}[n-i+2,n-1]$ 的最长公共前缀的长度。类似地,如果最长回文子串的长度是一个偶数,且其中心位置是 i,则其长度是 $s[i+1,n-1]$ 与 $s^{\sim}[n-i+1,n-1]$ 的最长公共前缀的长度。用一次线性扫描就可以找到最长回文子串。首先将输入字符串 s 变换为新字符串 t。

```
1  private static String change(String s)
2  {//字符串变换
3    String r=s;
4    r=new StringBuffer(s).reverse().toString();
5    String t=s+"1"+r+"0";
6    return t;
7  }
```

然后计算新字符串 t 的后缀数组 sa 和最长公共前缀数组 lcp,并建立一个类 rmq 来支持 t 的最长公共扩展查询。

```
1  private int lce(int l,int r)
2  {//最长公共扩展查询
3    return rq.rmq(Math.min(rank[l],rank[r]),Math.max(rank[l],rank[r])-1);
4  }
```

借助于 t 的最长公共扩展查询,对输入字符串做一次线性扫描就可以找到它的最长回文子串。

```
1  public int palin(String s)
2  {//最长回文子串
3    int n=s.length();
4    m=2*n+2;
5    String text=change(s);
6    SuffixDC3 suf=new SuffixDC3(text);
7    sa=suf.sa;
8    rank=suf.rank;
9    lcp=suf.lcp;
10    rq=new RMQ(m,lcp);
11    int ans=1,pos=0;
12    for(int i=0;i<n;i++){
13      int cp=lce(i,2*n-i);
14      if(cp*2-1>ans)
15      {ans=cp*2-1;pos=i-cp+1;}
16      cp=lce(i,2*n-i+1);
17      if(cp*2>ans)
18      {ans=cp*2;pos=i-cp;}
19    }
20    String t=text.substring(pos,Math.min(m,pos+ans));
```

```
21     System.out.println(ans);
22     System.out.println(t);
23     return ans;
24  }
```

由于计算新字符串 t 的后缀数组和最长公共前缀数组，并建立一个类 rmq 需要的计算时间是 $O(n)$。t 的最长公共扩展查询 lce 的响应时间是 $O(1)$。所以，一次线性扫描需要的计算时间是 $O(n)$。由此可见，整个算法需要的计算时间是 $O(n)$。

算法实现题 9-4　相似基因序列性问题

★ 问题描述

最长公共子序列问题是生物信息学中序列比对问题的一个特例。这类问题在分子生物学和模式识别中有着广泛的应用。其中，最主要的应用是所测量的基因序列的相似性。在演化分子生物学的研究中发现，某个重要的 DNA 序列片段常常出现在不同的物种中。在测量基因序列的相似性时，如果需要特别关注一个具体的 DNA 序列片段，就要考查带有子串排斥约束的最长公共子序列问题。这个问题可以具体表述如下：给定两个长度分别为 n 和 m 的序列 $x[0..n-1]$ 和 $y[0..m-1]$，以及一个长度为 p 的约束字符串 $s[0..p-1]$。带有子串排斥约束的最长公共子序列问题，就是要找出 x 和 y 的不包含 s 为其子串的最长公共子序列。例如，如果给定的序列 x 和 y 分别为 $x=$AATGCCTAGGC，$y=$CGATCTGGAC，字符串 $s=$TG 时，子序列 ATCTGGC 是 x 和 y 的一个无约束的最长公共子序列，而不包含 s 为其子串的最长公共子序列是 ATCGGC。

★ 算法设计

设计一个算法，找出给定序列 x 和 y 的不包含 s 为其子串的最长公共子序列。

★ 数据输入

由文件 input.txt 提供输入数据。文件的 3 行分别给出序列 x 和 y 以及约束字符串 s。

★ 结果输出

将计算出的 x 和 y 的不包含 s 为其子串的最长公共子序列的长度输出到文件 output.txt 中。

输入文件示例	输出文件示例
AATGCCTAGGC CGATCTGGAC TG	6

分析与解答

按照主教材中式(9.29)，可以设计求序列 x 和 y 的不包含 s 为其子串的最长公共子序列的长度的算法如下。首先为了有效计算函数 σ，对字符集中每个字符 ch$\in\Sigma$ 和 $1\leqslant k\leqslant p$，预先计算 $\sigma(s[1..k]\text{ch})$，并且保存在表 π 中。

```
1  private int cp(int i,int j)
2  {//预处理
3     if(i>0 && pi[i][j]==0) pi[i][j]=cp(kmp[i],j);
```

```
4     return pi[i][j];
5  }
```

用预先计算好的表 π 就可以在 $O(1)$ 时间内计算 $\sigma(s[1..k]ch)$ 的值。

```
1  private int r(char ch,int k)
2  {// 计算σ值
3     return pi[k][di(ch)];
4  }
```

```
1  public int dyna()
2  {//动态规划算法
3     for(int i=n;i>0;i--)
4        for(int j=m;j>0;j--)
5        for(int k=0;k<p;k++)
6           if(x[i]==y[j]){
7           f[i][j][k]=f[i+1][j+1][k];
8           int q=r(x[i],k);
9           if(q<p && f[i][j][k]<1+f[i+1][j+1][q])
10             f[i][j][k]=1+f[i+1][j+1][q];
11          }
12          else f[i][j][k]=Math.max(f[i+1][j][k],f[i][j+1][k]);
13    return f[1][1][0];
14 }
```

算法实现题 9-5　计算机病毒问题

★ 问题描述

计算机病毒是黑客在计算机程序中插入的破坏计算机功能或者数据的一组计算机指令或程序代码。计算机病毒不仅能影响计算机使用,还能自我复制。就像生物病毒一样,计算机病毒具有自我繁殖、互相传染和激活再生等生物病毒特征。计算机病毒的独特复制能力,使它们能够快速蔓延,又常常难以根除。计算机病毒能把自身附着在各种类型的文件上,当文件被复制或者从一个用户传送到另一个用户时,它们就随同文件一起蔓延开来。杀除计算机病毒的一个有效方法是找出特定计算机病毒的代码特征。对于给定的带有某种病毒的程序代码段集合,通过寻找程序代码段集合中所包含的公共特征,就可以快速地确定计算机病毒的代码特征。

★ 算法设计

给定带有某种病毒的程序代码段集合,寻找程序代码段集合中每个代码段都包含的最长字符串。

★ 数据输入

由文件 input.txt 提供输入数据。文件第 1 行有 1 个正整数 n,$1\leqslant n\leqslant 100$,表示程序代码段集合中代码段数。接下来的 n 行中,每行是一个程序代码段。每个程序代码段已经转换成由英文大小写字母组成的长度不超过 1000 的字符串。

★ 结果输出

将找到的程序代码段集合中最长公共字符串输出到文件 output. txt 中。

文件的第 1 行输出最长公共字符串的长度。文件的第 2 行输出最长公共字符串。

输入文件示例	输出文件示例
3	6
abcdefgi	bcdefg
cbcdefghc	
cdefgibcdefghe	

分析与解答：

此题要求给定字符串集合的最长公共子串。借助于后缀数组 sa 和最长公共前缀数组 lcp 所设计的算法如下：

```
1   public int lcs()
2   {//最长公共子串
3     readin();
4     build();
5     int max=search();
6     System.out.println(max);
7     if(max>0)
8       for(int j=0;j<max;j++) System.out.print(r.charAt(ans+j));
9     System.out.println();
10     return max;
11  }
```

在上述算法的第 3 行读入 n 个字符串，并将它们用不在输入串中出现的 n 个不同字符连接成一个新字符串 r。

```
1   public void readin()
2   {//读入 n 个字符串
3     Scanner sc=new Scanner(System.in);
4     try{sc=new Scanner(new FileInputStream("lcs.in"));}catch(Exception e){}
5     n=sc.nextInt();
6     r=new String();
7     idx=new int[maxn];
8     lab=new int[101];
9     for(int i=1;i<=n;i++){
10      String s=sc.next();
11      int k=s.length();
12      if(k<up)up=k;
13      r+=s;
14      for(int j=0;j<k;j++)idx[j+len]=i;
15      r+="0";
16      idx[len+k]=0;len+=k+1;
17    }
```

```
18    len   ;
19  }
```

在算法 lcs 中第 4 行的 build 建立新字符串 r 的后缀数组 sa 和最长公共前缀数组 lcp。

```
1  private void build()
2  {//建立后缀数组
3     SuffixDC3 suf=new SuffixDC3(r);
4     sa=suf.sa;
5     lcp=suf.lcp;
6     lcp[len]=-1;
7  }
```

在算法 lcs 中第 5 行的 search 利用后缀数组 sa 和最长公共前缀数组 lcp 做二分搜索来寻找最长公共子串。

```
1  private int search()
2  {//搜索公共子串
3     int first=1,last=up;
4     while(first<=last){
5        int mid=(first+last)>>1;
6        if(check(mid))first=mid+1;
7        else last=mid-1;
8     }
9     return last;
10  }
```

其中,check(mid)用于判断 r 中是否存在长度为 mid 的公共子串。

```
1  private boolean check(int mid)
2  {//长度为 mid 的公共子串
3     int j,t,s;
4     for(int i=0;i<len;i=j+1){
5        for(;lcp[i]<mid && i<=len;i++);
6        for(j=i;lcp[j]>=mid;j++);
7        if(j-i+1<n) continue;
8        cnt++;s=0;
9        for(int k=i;k<=j;k++)
10        if((t=idx[sa[k]])!=0 && lab[t]!=cnt){lab[t]=cnt;s++;}
11        if(s==n){ans=sa[i];return true;}
12     }
13    return false;
14  }
```

如果输入字符串集合中所有字符串长度总和为 l,其中最短字符串长度为 m,则上述算法需要的计算时间为 $O(l\log m)$。建立后缀数组 sa 和最长公共前缀数组 lcp 需要 $O(l)$ 时间,算法 check 需要 $O(l)$ 时间,二分搜索算法 search 最多调用 check 算法 $\log m$ 次。

算法实现题 9-6　带有子串包含约束的最长公共子序列问题

★ **问题描述**

给定两个长度分别为 n 和 m 的序列 $x[0..n-1]$ 和 $y[0..m-1]$，以及一个长度为 p 的约束字符串 $s[0..p-1]$。带有子串包含约束的最长公共子序列问题，就是要找出 x 和 y 的包含 s 为其子串的最长公共子序列。例如，如果给定的序列 x 和 y 分别为 $x=$ AATGCCTAGGC，$y=$ CGATCTGGAC，字符串 $s=$ GTA 时，子序列 ATCTGGC 是 x 和 y 的一个无约束的最长公共子序列，而包含 s 为其子串的最长公共子序列是 GTAC。

★ **算法设计**

设计一个算法，找出给定序列 x 和 y 的包含 s 为其子串的最长公共子序列。

★ **数据输入**

由文件 input.txt 提供输入数据。文件的 3 行分别给出序列 x 和 y 以及约束字符串 s。

★ **结果输出**

将计算出的 x 和 y 的包含 s 为其子串的最长公共子序列的长度输出到文件 output.txt 中。

输入文件示例	输出文件示例
AATGCCTAGGC	4
CGATCTGGAC	
GTA	

分析与解答：

按照主教材中式(9.24)和式(9.25)，可以设计求 x 和 y 的包含 s 为其子串的最长公共子序列的长度的算法如下：

```
1   public int comp()
2   {//带有子串包含约束的最长公共子序列
3     suffix();
4     lcsr();
5     int i1,j1,tmp=0;
6     for(int i=1;i<=n;i++)
7       for(int j=1;j<=m;j++){
8       int sum=f[i][j][p];
9       if(i<n && j<m) sum+=g[i+1][j+1];
10       if(tmp<sum) {tmp=sum;i1=i;j1=j;}
11    }
12    return tmp;
13  }
```

其中，suffix 用于计算 $f(i,j,k)$，lcsr 用于计算 $g(i,j)$。

```
1   private int suffix()
2   {//计算 f(i,j,k)
3     for(int i=0;i<=n;i++) f[i][0][0]=0;
4     for(int j=0;j<=m;j++) f[0][j][0]=0;
```

```
5      for(int k=1;k<=p;k++){
6        for(int i=0;i<=n;i++)f[i][0][k]=Integer.MIN_VALUE;
7        for(int j=0;j<=m;j++)f[0][j][k]=Integer.MIN_VALUE;
8      }
9      for(int i=1;i<=n;i++)
10         for(int j=1;j<=m;j++)
11           for(int k=0;k<=p;k++){
12           int d=f[i][j][k];
13           if(x[i-1]!=y[j-1]) d=Math.max(f[i][j-1][k],f[i-1][j][k]);
14           else{
15           if (k==0 && x[i-1]==y[j-1]) d=f[i-1][j-1][k]+1;
16           if (k>0 && x[i-1]!=z[k-1]) d=f[i-1][j-1][k];
17           else if(k>0 && x[i-1]==z[k-1]) d=1+f[i-1][j-1][k-1];
18           }
19           f[i][j][k]=d;
20         }
21     return Math.max(-1,f[n][m][p]);
22   }
```

```
1  private int lcsr()
2  {//计算 g(i,j,k)
3    g[n+1][m+1]=0;
4    for (int i=1;i<= n; i++) g[i][n+1]=0;
5    for (int i=1;i<= m; i++) g[m+1][i]=0;
6    for(int i=n;i>0;i--)
7      for(int j=m;j>0;j--)
8      if (x[i-1]==y[j-1]) g[i][j]=g[i+1][j+1]+1;
9      else if (g[i+1][j]>g[i][j+1]) g[i][j]=g[i+1][j];
10      else g[i][j]=g[i][j+1];
11      return g[1][1];
12  }
```

算法实现题 9-7　多子串排斥约束的最长公共子序列问题

★ 问题描述

给定两个长度分别为 n 和 m 的序列 $x[0..n-1]$ 和 $y[0..m-1]$，以及 d 个约束字符串 $s_1,s_2,\cdots,s_d$。多子串排斥约束的最长公共子序列问题，就是要找出 x 和 y 的不含 $s_1,s_2,\cdots,s_d$ 为其子串的最长公共子序列。

★ 算法设计

设计一个算法，找出给定序列 x 和 y 的不含 $s_1,s_2,\cdots,s_d$ 为其子串的最长公共子序列。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行中给出正整数 d，表示约束字符串个数。接下来的 2 行分别给出序列 x 和 y。最后 d 行的每一行给出一个约束字符串。

★ 结果输出

将计算出的 x 和 y 的不含 $s_1,s_2,\cdots,s_d$ 为其子串的最长公共子序列输出到文件 output.

txt 中。文件的第 1 行输出最长公共子序列,第 2 行输出最长公共子序列的长度。

输入文件示例	输出文件示例
4	CCC
AATGCCTAGGC	3
TG	
A	
G	
TC	

分析与解答:

给定输入序列 $x[0..n-1]$ 和 $y[0..m-1]$,以及 d 个约束字符串 $s_1,s_2,\cdots,s_d$,其总长度是 r。问题求解目标是找到 x 和 y 的最长公共子序列,不含任何 $s_1,s_2,\cdots,s_d$ 为其子串。这个问题实际上是有子串排除约束的最长公共子序列在多子串情况下的推广,算法思想是类似的。在多子串的情形,需要用到 AC 自动机。首先建立字符串约束集 $s_1,s_2,\cdots,s_d$ 的 AC 自动机 T。设 T 中非叶结点编号为 $0,1,\cdots,t-1$。根结点编号为 0。在 AC 自动机 T 中,从结点根 0 到任一状态结点 state 的路径上各边的标号字符连接组成的字符串,即结点 state 的标号为 $\alpha(\text{state})$。对任一字符串 q,它在 AC 自动机 T 中的最长后缀结点记为 $\sigma(q)$,即

$$|\alpha(\sigma(q))| = \max_{0 \leqslant i < t}\{|\alpha(i)| \mid \alpha(i) \quad q\}$$

设 $Z(i,j,k)$ 是 $x[0..i]$ 和 $y[0..j]$ 的不含任何 $s_1,s_2,\cdots,s_d$ 中字符串为其子串的最长公共子序列组成的集合,且对任一 $z \in Z(i,j,k)$ 有 $\sigma(z)=k$,$0 \leqslant i \leqslant n-1$,$0 \leqslant j \leqslant m-1$,$0 \leqslant k < t$。

$Z(i,j,k)$ 中任一最长公共子序列的长度记为 $f(i,j,k)$。

如果能有效计算出 $f(i,j,k)$,则 x 和 y 的不含任何 $s_1,s_2,\cdots,s_d$ 为其子串的最长公共子序列的长度就是 $\max\limits_{0 \leqslant k < t}\{f(n,m,k)\}$。

用动态规划算法计算 $f(i,j,k)$ 的递归式如下:

$$f(i,j,k) = \begin{cases} \max\{f(i-1,j,k), f(i,j-1,k)\} & x_i \neq y_j \\ \max\{f(i-1,j-1,k), 1+\beta(i,j,k)\} & x_i = y_j \end{cases}$$

其中,

$$\beta(i,j,k) = \max_{0 \leqslant q < t}\{f(i-1,j-1,q) \mid \sigma(\alpha(q)x_i) = k\}$$

用字符串约束集 $s_1,s_2,\cdots,s_d$ 的 AC 自动机 T,可以在 $O(1)$ 时间内计算 $\sigma(q)$。因此,上述动态规划算法所需计算时间为 $O(nmr)$。

第 10 章 算法优化策略

习题 10-1 算法 obst 的正确性

证明第 3 章中解最优二叉搜索树问题的 $O(n^2)$时间算法 obst 的正确性。

分析与解答：

第 3 章解最优二叉搜索树问题的 $O(n^2)$时间算法 obst 中，

$$w_{ij} = a_{i-1} + b_i + \cdots + b_j + a_j, \quad 1 \leqslant i \leqslant j \leqslant n$$

对于任意的 $i \leqslant i' < j \leqslant j'$，显然有

$$w(i,j) + w(i',j') = w(i',j) + w(i,j')$$

可见 w 是满足四边形不等式的单调函数，从而函数 $s(i,j)$单调。由此可得

$$\min_{i \leqslant k \leqslant j}\{m(i,k-1) + m(k+1,j)\} = \min_{s[i][j-1] \leqslant k \leqslant s[i+1][j]}\{m(i,k-1) + m(k+1,j)\}$$

习题 10-2 矩阵连乘问题的 $O(n^2)$时间算法

试设计解矩阵连乘问题的 $O(n^2)$时间算法。

分析与解答：

(1) 矩阵连乘积的最优计算次序问题是凸多边形最优三角剖分问题的特殊情形。对于给定的矩阵链 $\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_n$，定义与之相应的凸$(n+1)$边形 $P=\{v_0,v_1,\cdots,v_n\}$，使得矩阵 $\mathbf{A}_i$ 与凸多边形的边 $v_{i-1}v_i$ 一一对应。若矩阵 $\mathbf{A}_i$ 的维数为 $p_{i-1}\times p_i$，$i=1,2,\cdots,n$，则定义凸多边形的顶点的权值为 $w_i=w(v_i)=p_i$；三角形 $v_iv_jv_k$ 上的权函数值为：$w(v_iv_jv_k)=p_ip_jp_k$。依此权函数的定义，凸多边形 P 的最优三角剖分所对应的语法树给出矩阵链 $\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_n$的最优完全加括号方式。

(2) 为便于叙述，将凸多边形 P 的顶点重新编号，使得 $w_0\leqslant w_1\leqslant\cdots\leqslant w_n$。

考查 3 个矩阵连乘的情形。此时相应的多边形 P 是一个四边形。容易看出，当 w_0 与 w_1 不相邻时，弦 w_0w_1 给出 P 的最优三角剖分；类似地，当 w_0 与 w_2 不相邻时，弦 w_0w_2 给出 P 的最优三角剖分；当 w_0 与 w_3 不相邻时，弦 w_1w_2 和弦 w_0w_3 都有可能给出 P 的最优三角剖分。这个简单结论可以推广到一般情形。

设 P 是一个凸$(n+1)$边形。它的 $n+1$ 个顶点的权为 $w_0\leqslant w_1\leqslant\cdots\leqslant w_n$。则存在 P 的最优三角剖分 π，使得

① $w_0w_1\in\pi$，且 $w_0w_2\in\pi$。

② 若 w_0w_1 与 w_0w_2 均为 P 的边，则 $w_0w_3\in\pi$，或 $w_1w_2\in\pi$。

据此可以设计求 P 的最优三角剖分 π 的递归算法如下：

```
void Partition(P)
{
    if(size(P)==1 ‖ size(P)==2) return ∅;
    else if(size(P)==3) return P;
    else if(w0 与 w1 不相邻) return Partition(P12)∪Partition(P21);
    else if(w0 与 w2 不相邻) return Partition(P13)∪Partition(P31);
    else return min(Partition(P23)∪Partition(P32),Partition(P14)∪Partition(P31));
}
```

其中，P_{ij} 表示多边形 P 在 w_i 与 w_j 之间按照顺时针方向列出的顶点组成的子多边形。

由于算法 Partition 最后的 else 语句可能产生 2 个顶点数为 $O(n)$的子多边形，Partition 需要指数时间。

(3) $O(n^2)$时间算法描述。

事实上，上述算法中只有 $O(n^2)$个不同的子多边形。这可以用多边形 P 的水平弦来刻画。对于凸多边形 P 中任一弦 $r=w_iw_j$，设 $P(r)$是在凸多边形 P 上依顺时针方向从顶点 w_i 到顶点 w_j 所经历的所有顶点组成的子多边形。若对 $P(r)$中所有顶点 w_k 均有，$w_k \geq \max\{w_i, w_j\}$，则称 r 是凸多边形 P 的一条水平弦，$P(r)$是 P 的以水平弦 r 为界的上子多边形。

易知，凸多边形中的水平弦互不相交。因此，凸多边形 P 最多只有 $O(n)$条水平弦。

设 r_1 和 r_2 是凸多边形 P 中的 2 条水平弦。当 $P(r_1) \subseteq P(r_2)$时，称 r_1 小于 r_2，并记为 $r_1 < r_2$。亦称 r_2 大于 r_1，并记为 $r_2 > r_1$。当 $r_1 < r_2$ 或 $r_1 > r_2$ 时，称水平弦 r_1 和 r_2 是可比较的，否则称 r_1 和 r_2 是互相独立的水平弦。水平弦之间的<关系和>关系均为偏序关系。凸多边形 P 的水平弦之间的偏序关系图是一个由 2 棵二叉树组成的森林，称为 P 的水平弦森林，如图 10-1 所示。P 的水平弦森林中任一非叶结点 $r=w_iw_j$ 恰有 2 个儿子结点 w_iw_k 和 w_kw_j，其中 w_k 是 $P(r)$中除 w_i 和 w_j 外的最小权顶点。当 $w_i < w_j$ 时，记 $w_iw_k = \min \operatorname{son}(r)$，$w_kw_j = \max \operatorname{son}(r)$。

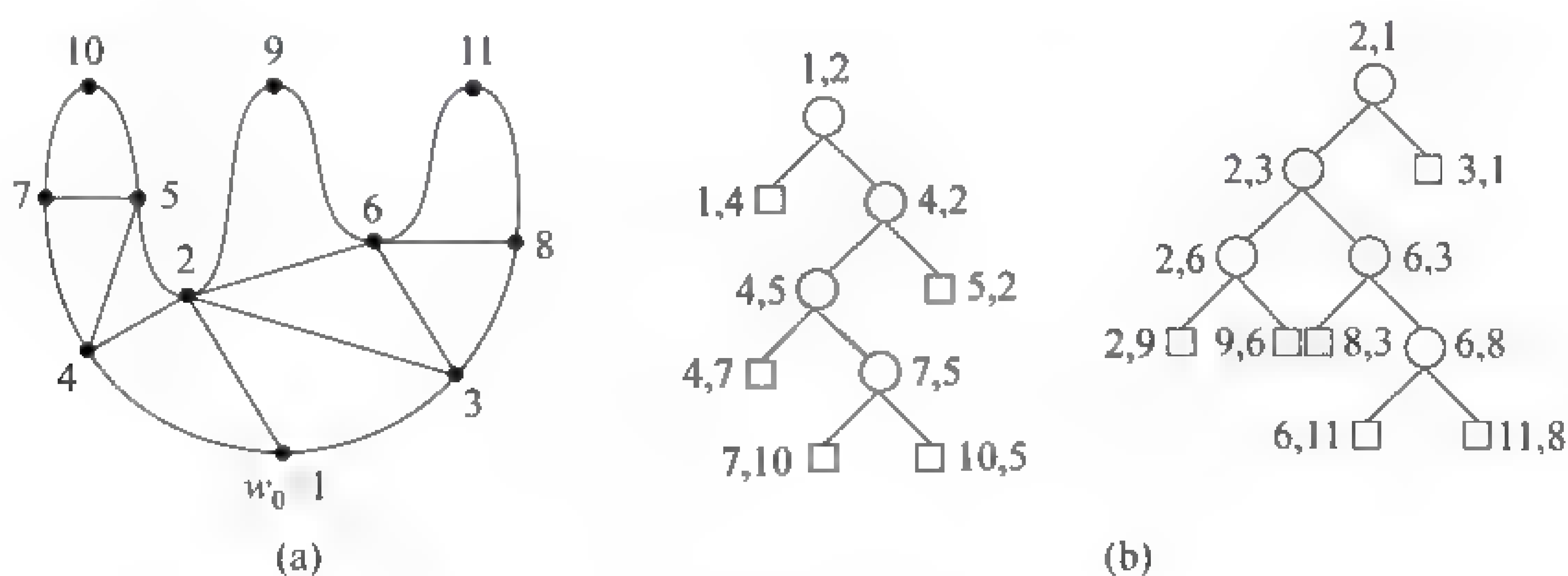


图 10-1 凸多边形的水平弦森林

下面的算法以后序遍历方式在 $O(n)$时间内找出给定凸多边形 P 的所有水平弦，并将它们按水平弦偏序关系<排列。

```
void chords(P)
{
    w=w0;
    do{
```

```
            S.push(w);
            w=next(w);
            while(S.top>w){
                t=S.pop();
                output(S.top,t);
                output(t,w);
            }
        }while(w!=w0)
    }
```

设 $r=w_iw_j$ 是 P 的一条水平弦。对于 $k\leqslant \min\{i,j\}$，定义 P 的子多边形 $P_{ij}\cup w_iw_jw_k$ 为 P 的一个锥，记为(r,w_k)。

对于锥 $Q=(r,w_k)$，在算法 Partition(Q)中出现的子多边形是一个三角形或是 Q 的子锥。因此可用动态规划算法将每条水平弦所相应的锥的最优剖分以自底向上的方式计算出来。多边形 P 最多有 $O(n)$ 条水平弦，每条水平弦最多对应 n 个锥，因此最多有 $O(n^2)$个锥。

$O(n^2)$时间动态规划算法 DP-Partition 描述如下：

```
void DP-Partition(P)
{
    for(b=wiwj∈B){// B是算法chords(P)输出的水平弦集合,i<j
        if(leaf(b)){
            for(k=0;k<=i;k++){
                Q=(b,wk);
                if(k=i)Partition(Q)=∅;
                else Partition(Q)=Q;
            }
        }
        else{
            for(k=0;k<=i;k++){
                Q=(b,wk);
                if(k=i)Partition(Q)=Partition((minsob(b),wi))∪Partition((maxsob(b),wi));
                else {
                    P1=Partition((b,wi))∪wiwjwk;
                    P2=Partition((minsob(b),wk))∪Partition((maxsob(b),wk));
                    Partition(Q)=min{P1,P2};
                }
            }
        }
    }
    Partition(P)=Partition(P12)∪Partition(P21);
}
```

(4) 算法描述。

用一个类 Arc 表示水平弦。

```
public static class Arc
```

```
{
    int v1,v2,son1,son2;
```

其中,v1 和 v2 分别是水平弦的 2 个端点;son1 和 son2 分别是水平弦在水平弦树中的 2 个儿子结点。

input 读入数据并初始化。

```
static void input()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    p=new int [n+1];
    r=new int [n+1];
    for (int i=0;i<=n;i++) p[i]=keyboard.readInt();
    int i=mini(p,n);
    shift(p,i,n);
    rank(p,n+1,r);
    chords();
}
```

上述算法中的 mini 求最小维数下标。

```
static int mini(int []p,int n)
{
    int j=0;
    for(int i=1;i<=n;i++)if(p[i]<p[j])j=i;
    return j;
}
```

shift 平移使 $p[0]$ 为最小维数。

```
static void shift(int []p,int i,int n){Exchange.exch(p,n+1,i);}
```

rank 计算各维数的秩。

```
public static void rank(int []a, int n, int []r)
{
    for (int i=0;i<n;i++) r[i]=0;
    for (int i=1;i<n;i++)
        for (int j=0;j<i;j++)
            if (a[j]<=a[i]) r[i]++;
            else r[j]++;
}
```

chords 计算多边形 P 的所有水平弦,并建立相应的水平弦森林。

```
static void chords()
{
    int t,j=0;
```

```
        ArrayStack S=new ArrayStack(2 * n);
        a=new Arc[2 * n];
        for(int i=0;i<2 * n;i++)a[i]=new Arc();
        for (int i=0;i<=n;i++){
            S.push(new Integer(i));
            while (larger(((Integer)S.peek()).intValue(),(i+1)%(n+1))){
                t=((Integer)S.pop()).intValue();
                a[j].son1=-1;a[j].son2=-1;
                a[j].v1=((Integer)S.peek()).intValue();a[j++].v2=t;
                a[j].son1=-1;a[j].son2=-1;
                a[j].v1=t;a[j++].v2=(i+1)%(n+1);
            }
        }
        while (!S.empty()) t=((Integer)S.pop()).intValue();
        S.push(new Integer(0));
        for (int i=1;i<j/2;i++){
            t=1;
            while (t>0 && !S.empty()){
                int k=((Integer)S.peek()).intValue();
                t=son(k,i);
                if (t>0){a[t].son1=2 * k;a[t].son2=2 * k+1;k=((Integer)S.pop()).intValue();}
            }
            S.push(new Integer(i));
        }
    }
```

其中,函数 larger(x,y) 用于判断顶点 x 和 y 的大小。函数 son(k,i)判断弦 $2k$ 和 $2k+1$ 是否为弦 $2i$ 或 $2i+1$ 的儿子结点。

```
    static boolean larger(int x, int y){ return (p[x]>p[y]) || (p[x]==p[y])&&(x>y);}

    static int son(int k,int i)
    {
        int x=a[2 * k].v1;
        int y=a[2 * k+1].v2;
        int t=0;
        if ((a[2 * i].v1==x)&&(a[2 * i].v2==y)) t=2 * i;
        if ((a[2 * i+1].v1==x)&&(a[2 * i+1].v2==y)) t=2 * i+1;
        return t;
    }
```

matrixChain 实现前面描述的 $O(n^2)$时间动态规划算法 DP-Partition。

```
    static int matrixChain()
    {
        if (n<2) return 0;
        chords();
        opt=new int[2 * n][n+1];
        for(int i=0;i<2 * n;i++){
```

```
            int ii=a[i].v1,jj=a[i].v2;
            if(r[ii]>r[jj]){int tmp=ii;ii=jj;jj=tmp;}
            if(leaf(a[i])){
                for(int k=0;k<=n;k++){
                    if(r[k]==r[ii])opt[i][k]=0;
                    if(r[k]<r[ii])opt[i][k]=p[ii]*p[jj]*p[k];
                }
            }
            else{
                int s1=a[i].son1,s2=a[i].son2;
                int t1=opt[s1][ii]+opt[s2][ii];
                for(int k=0;k<=n;k++){
                    if(r[k]==r[ii])opt[i][k]=t1;
                    if(r[k]<r[ii]){
                        opt[i][k]=t1+p[ii]*p[jj]*p[k];
                        int tmp=opt[s1][k]+opt[s2][k];
                        if(tmp<opt[i][k]) opt[i][k]=tmp;
                    }
                }
            }
        }
        return (opt[2*n-1][0]+opt[2*n-2][0]);
    }
```

其中,leaf 判断叶结点。

```
static boolean leaf(Arc e){ return (e.son1<0 && e.son2<0);}
```

习题 10-3　货物储运问题的费用

货物储运问题中,当合并 $a[i]$和 $a[j]$所需费用不是 $a[i]+a[j]$,而是别的简单函数,如 $a[i]\times a[j]$或$|a[i]-a[j]|$时,如何设计有效算法?

分析与解答:

对于可加费用函数而言,与教材中的设计方法类似。

习题 10-4　Garsia 算法

设计实现货物储运问题的另一个稍不同的最优算法如图 10-2 所示。该算法与 10.3 节所述算法的区别在于组合阶段的策略稍有不同。标记层序阶段和重组阶段完全相同。试设计实现上述思想的 $O(n\log n)$时间算法。

分析与解答:

所述算法是著名的 Garsia 算法。算法仍分 3 个阶段。

1) 组合阶段

将给定的 n 个数依序从左到右排列 a_0, a_1,…,a_{n-1}。

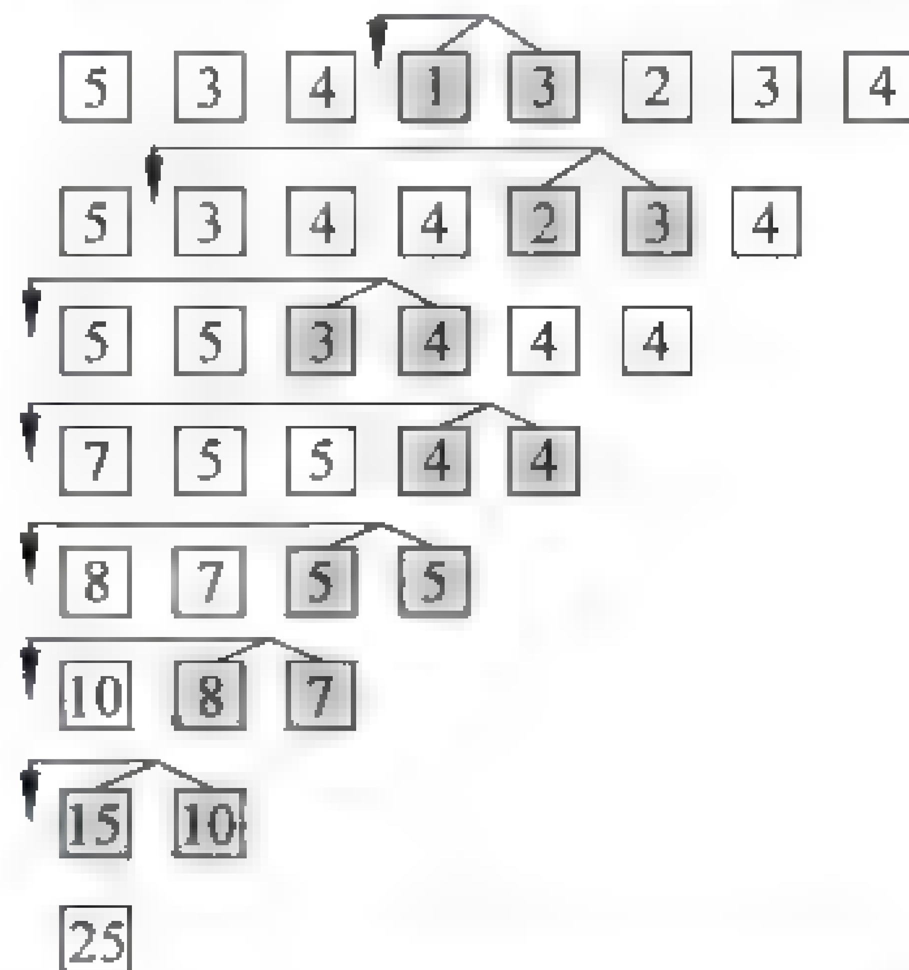


图 10-2　货物储运问题的最优算法

首先找到使 $a_{k-1}<a_{k+1}$的最小下标 k,然后找使得

$j<k$ 且 $a_{j-1} \geq a_{k-1}+a_k$ 的最大下标 j，接着删去 a_{k-1} 和 a_k 并将其和 $a_{k-1}+a_k$ 插入 a_{j-1} 之后。重复上述过程，直至序列中只剩下 1 个结点。

2) 标记层序阶段

将第一阶段结束后留下的唯一结点标记为第 0 层结点。然后以与第 1 阶段相反的组合顺序标记其余结点的层序。

3) 重组阶段

根据标记层序阶段计算出的各结点的层序，按下述规则重组。

结点 a_i 和 a_j 重组为新结点应满足如下要求：

(1) a_i 和 a_j 在当前序列中相邻。

(2) a_i 和 a_j 均为当前序列中最大层序结点。

按照上述思想设计的算法实现如下：

```
static int n,m,t,sum=0;
static int []w;
static int []l;
static int []r;
static int []d;
static int []q;
static int []v;
public static void main(String [] args)
{
    init();
    phase1();
    phase2();
    phase3();
    System.out.println(sum);
}
```

init 读入数据并初始化。

```
static void init()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    w=new int[2*n+1];
    l=new int[2*n+1];
    r=new int[2*n+1];
    d=new int[2*n+1];
    q=new int[2*n+1];
    v=new int[2*n+1];
    for(int j=0;j<n;j++){w[j]=keyboard.readInt();l[j]=r[j]=-1;}
}
```

phase1 实现组合阶段功能。

```
static void phase1()
```

```
{
    m=n;t=1;
    q[0]=Integer.MAX_VALUE;
    q[1]=w[0];v[1]=0;
    for(int k=1;k<n;k++){
        while(w[k]>=q[t-1])combine();
        t++;q[t]=w[k];v[t]=k;
    }
    while(t>1)combine();
}
```

combine 实现具体的结点合并。

```
static void combine()
{
    int j,k,x;
    k=t;
    do{
        m++;t--;
        l[m]=v[k-1];r[m]=v[k];
        w[m]=x=q[k-1]+q[k];
        for(j=k;j<=t;j++){q[j]=q[j+1];v[j]=v[j+1];}
        for(j=k-2;q[j]<x;j--){q[j+1]=q[j];v[j+1]=v[j];}
        q[j+1]=x;v[j+1]=m;k=j;
    }while (k> 0&&q[k-1]<=x);
}
```

phase2 实现标记层序阶段功能。

```
static void phase2(){mark(v[1],0);}
static void mark(int k,int p)
{
    d[k]=p;
    if(l[k]>=0)mark(l[k],p+1);
    if(r[k]>=0)mark(r[k],p+1);
}
```

phase3 实现重组阶段功能。

```
static void phase3(){t=0;m=2*n-2;build(1);}
static void build(int b)
{
    int j=m;
    if(d[t]==b)l[j]=t++;
    else{m--;l[j]=m;build(b+1);}
    if(d[t]==b)r[j]=t++;
    else{m--;r[j]=m;build(b+1);}
    w[j]=w[l[j]]+w[r[j]];
```

```
sum += w[j];
```

算法实现题 10-1　货物储运问题

★ 问题描述

在一个铁路沿线顺序存放着 n 堆装满货物的集装箱。货物储运公司要将集装箱有次序地集中成一堆。规定每次只能选相邻的两堆集装箱合并成新的一堆,所需的运输费用与新的一堆中集装箱数成正比。给定各堆的集装箱数,试制定一个运输方案,使总运输费用最少。

设 n 堆货物从左到右编号为 $1,2,\cdots,n$。各堆货物集装箱数为 $a[1:n]$。

★ 算法设计

对于给定 n 堆货物,计算合并成一堆的最少运输费用。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n,表示有 n 堆货物。第 2 行有 n 个数,分别表示每堆货物的集装箱数。

★ 结果输出

将计算出的最少运输费用输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4	43
4 4 5 9	

分析与解答:

$O(n\log n)$时间算法见主教材。

算法实现题 10-2　石子合并问题

★ 问题描述

在一个圆形操场的四周摆放着 n 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的两堆石子合并成新的一堆,并将新的一堆石子数记为该次合并的得分。试设计一个算法,计算出将 n 堆石子合并成一堆的最小得分。

★ 算法设计

对于给定 n 堆石子,计算合并成一堆的最小得分。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n,表示有 n 堆石子。第 2 行有 n 个数,分别表示每堆石子的个数。

★ 结果输出

将计算出的最小得分输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4	54
4 4 5 9	

分析与解答：

圆排列合并问题与直线排列合并问题的不同之处在于改变了 $a[1]$ 与 $a[n]$ 的相邻性。在直线排列的情形，长度为 n 的合并序列只有 1 种，即 $1,2,\cdots,n$。而在圆排列的情形，长度为 n 的合并序列共有以下 n 种：

$1,2,\cdots,n$；

$2,3,\cdots,n,1$；

$3,4,\cdots,n,1,2$；

$n,1,2,\cdots,n-1$。

将圆环断开，拉成直线，如图 10-3 所示。

$$\underbrace{a[1],a[2],\cdots,a[n]}_{\text{实际的}},\ \underbrace{a[1],a[2],\cdots,a[n-1]}_{\text{延伸的}}$$

图 10-3　圆排列直线化

在这种扩充的直线排列下，圆排列的 n 种情形都能表示。设扩充出的 $n-1$ 个元素为 $a[n+i]$，$i=1\sim n-1$，则 $a[n+i]=a[i]$，$i=1\sim n-1$。这样就将 n 元圆排列合并问题转化为 $2n-1$ 元直线排列合并问题。

算法实现题 10-3　最大运输费用货物储运问题

★ 问题描述

在一个铁路沿线顺序存放着 n 堆装满货物的集装箱。货物储运公司要将集装箱有次序地集中成一堆。规定每次只能选相邻的两堆集装箱合并成新的一堆，所需的运输费用与新的一堆中集装箱数成正比。给定各堆的集装箱数，试确定使总运输费用最大的运输方案。

设 n 堆货物从左到右编号为 $1,2,\cdots,n$。各堆货物集装箱数为 $a[1:n]$。

★ 算法设计

对于给定 n 堆货物，计算合并成一堆的最大运输费用。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n，表示有 n 堆货物。第 2 行有 n 个数，分别表示每堆货物的集装箱数。

★ 结果输出

将计算出的最大运输费用输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4	43
4 4 5 9	

分析与解答：

1）最优子结构性质

对于 $a[1:n]$ 的一个最优合并方式，设其在 $a[k]$ 和 $a[k+1]$ 之间断开，则其合并方式为 $((a[1:k])(a[k+1:n]))$。容易看出，此时 $a[1:k]$ 和 $a[k+1:n]$ 的合并方式也是最优的，即该问题具有最优子结构性质。

2）递归关系

设合并 $a[i:j]$，$1\leqslant i\leqslant j\leqslant n$，所需的最大费用为 $m[i,j]$，则原问题的最优值为 $m[1,n]$。由最优子结构性质可知，

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i<k\leqslant j}\{m[i,k-1]+m[k,j]+\sum\limits_{t=i}^{j}a[t]\} & i<j\end{cases}$$

3) 算法改进

对于最大运输费用货物储运问题的递归式,可以进一步改进为

$$\max_{i<k\leqslant j}\{m(i,k-1)+m(k,j)\}=\max\{m(i,j-1),m(i+1,j)\},\quad 1\leqslant i<j\leqslant n$$

事实上,由算法的递归式可知,$m(i,j)=\max\limits_{i<k\leqslant j}\{m(i,k-1)+m(k,j)\}+\sum\limits_{t=i}^{j}a[t]$。

设 $i+1\leqslant p\leqslant j$,使 $m(i,j)=m(i,p-1)+m(p,j)+\sum\limits_{t=i}^{j}a[t]$。与此相应的合并方式可表示为$((a[i],\cdots,a[p-1])(a[p],\cdots,a[j]))$。

(1) 当 $p=i+1$ 或 $p=j$ 时,显然有$\max\limits_{i<k\leqslant j}\{m(i,k-1)+m(k,j)\}=\max\{m(i,j-1),m(i+1,j)\}$。

(2) 当 $i+1<p<j$ 时,设 $u=\sum\limits_{t=i}^{p-1}a[t]$,$v=\sum\limits_{t=p}^{j}a[t]$。下面分两种情形讨论。

① $u\geqslant v$ 的情形。

由于 $p<j$,故可设 $m(p,j)=m(p,r-1)+m(r,j)+v_1+v_2$。其中,$v_1=\sum\limits_{t=p}^{r-1}a[t]$,$v_2=\sum\limits_{t=r}^{j}a[t]$ 且 $v_1+v_2=v$。与此相应的合并方式可表示为

$$((a[i],\cdots,a[p-1])((a[p],\cdots,a[r-1])(a[r],\cdots,a[j])))$$

其相应的费用为

$$m(i,j)=m(i,p-1)+m(p,r-1)+m(r,j)+u+v+v_1+v_2$$

将这一合并方式适当改变如下:

$$(((a[i],\cdots,a[p-1])(a[p],\cdots,a[r-1]))(a[r],\cdots,a[j]))$$

与此合并方式相应的费用 $T(i,j)$为 $T(i,j)=m(i,p-1)+m(p,r-1)+m(r,j)+2u+2v_1+v_2$。

由于 $a[i]>0,1\leqslant i\leqslant n$,所以 $v_1,v_2>0$。由此可推出,

$T(i,j)-m(i,j)=u-v_2=u-v+v-v_2=u-v+v_1\geqslant v_1>0$,即 $T(i,j)>m(i,j)$。此为矛盾。

② $u<v$ 的情形。

与①的情形类似,可推出矛盾。

至此已证明,对于最大运输费用货物储运问题有

$$\max_{i<k\leqslant j}\{m(i,k-1)+m(k,j)\}=\max\{m(i,j-1),m(i+1,j)\},\quad 1\leqslant i<j\leqslant n$$

具体算法描述如下:

```
public static int maxsum (int a[ ])
{
    for (int i=2; i<=n; i++) a[i]=a[i]+a[i-1];
    for (int r=2; r<=n; r++)
```

```
        for (int i=1; i<=n-r+1; i++){
            int j=i+r-1;
            if (m[i+1][j]>m[i][j-1]) m[i][j]=m[i+1][j]+a[j]-a[i-1];
            else m[i][j]=m[i][j-1]+a[j]-a[i-1];
        }
    return m[1][n];
}
```

算法实现题 10-4　五边形问题

★ 问题描述

给定平面上 n 个点组成的集合 X，找出 X 中点所张成的周长最大的凸五边形。

★ 算法设计

对于给定的平面点集 X，设计一个算法，求 X 中点张成的周长最大的凸五边形。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示集合 X 中有 n 个点。接下来的 n 行中，每行有 2 个整数，分别表示点的 x 坐标和 y 坐标值。

★ 结果输出

将计算出的最大凸五边形的周长输出到文件 output.txt。输出结果保留 2 位小数，4 舍 5 入。

输入文件示例	输出文件示例
input.txt	output.txt
6	8.83
0 0	
0 2	
1 2	
1 3	
2 0	
2 2	

分析与解答：

(1) 先求 X 的凸壳 $\mathrm{conv}(X)=\{v_0,v_1,\cdots,v_{n-1}\}$。

(2) 计算凸壳各顶点间的距离 $\boldsymbol{D}=(d_{ij})$。

(3) 定义运算$\otimes$：$\boldsymbol{A}\otimes\boldsymbol{B}=(c_{ij})$，$c_{ij}=\max\{a_{ik}+b_{kj}\mid i\leqslant k\leqslant j\}$。

则最大周长的三角形的周长由矩阵 $\boldsymbol{D}+\boldsymbol{D}\odot\boldsymbol{D}$ 中最大元素值给出。同理可知，最大周长凸五边形的周长由矩阵 $\boldsymbol{D}+\boldsymbol{D}^2\odot\boldsymbol{D}^2$ 中最大元素值给出。一般情况下，最大周长凸 m 边形的周长由矩阵 $\boldsymbol{D}+\boldsymbol{D}^{m-1}$ 中最大元素值给出。

(4) 计算 $\boldsymbol{D}\odot\boldsymbol{D}$ 需要 $O(n^3)$ 计算时间，从而计算 $\boldsymbol{D}+\boldsymbol{D}^{m-1}$ 需要 $O(n^3\log m)$ 计算时间。

(5) $\boldsymbol{D}\odot\boldsymbol{D}$ 的计算式满足四边形不等式，故可将计算 $\boldsymbol{D}\odot\boldsymbol{D}$ 的计算时间减至 $O(n^2)$，从而计算 $\boldsymbol{D}+\boldsymbol{D}^{m-1}$ 需要 $O(n^2\log m)$ 计算时间。

具体算法实现如下。

input 读入初始数据，计算 X 的凸壳，以及距离矩阵 $\boldsymbol{D}=(d_{ij})$。

```
static void input()
{
    ReadStream keyboard=new ReadStream();
```

```
        n=keyboard.readInt();
        for(int i=0;i<n;i++){
            int x=keyboard.readInt();
            int y=keyboard.readInt();
            plist.SetVertex(x,y);
        }
        conv=new int[n][2];
        n=convexhull(conv);
        dism();
    }
```

其中,convexhull 计算 X 的凸壳,dism()计算距离矩阵。

```
    static void dism()
    {
        d=new double [n][n];
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)d[i][j]=dist(i,j);
    }

    static double dist(int i,int j)
    {
        double dd=Math.sqrt(Math.pow((conv[i][0]-conv[j][0]),2)+Math.pow((conv[i][1]-
                conv[j][1]),2));
        return dd;
    }
```

mult 计算 $z=x\otimes y$。

```
    static void mult(double [][]x,double [][]y,double [][]z)
    {
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++){
                z[i][j]=x[i][j]+y[j][j];
                for(int k=i;k<j;k++){
                    double tmp=x[i][k]+y[k][j];
                    if(tmp>z[i][j])z[i][j]=tmp;
                }
            }
    }
```

mult 需要 $O(n^3)$计算时间。利用四边形不等式,可将 $x\odot y$ 的计算时间减至$O(n^2)$。

```
    static void qmult(double [][]x,double [][]y,double [][]z)
    {
        int [][]s=new int[n][n];
        for (int i=0;i<n;i++)s[i][i]=0;
        for (int r=1;r<n;r++)
            for (int i=0;i<n-r;i++){
                int j=i+r;
```

```
            i1=s[i][j-1]>i?s[i][j-1]:i;
            j1=s[i+1][j]<j?s[i+1][j]:j;
         z[i][j]=x[i][i1]+y[i1][j];s[i][j]=i1;
         for (int k=i1+1;k<=j1;k++){
            double t=x[i][k]+y[k][j];
            if (t>z[i][j]){z[i][j]=t;s[i][j]=k;}
         }
      }
   for (int r=1;r<n;r++)
      for (int j=0;j<n-r;j++){
         int i=j+r,
            i1=s[i-1][j]>i?s[i-1][j]:i,
            j1=s[i][j+1]<j?s[i][j+1]:j;
         z[i][j]=x[i][i1]+y[i1][j];
         s[i][j]=i1;
         for (int k=i1+1;k<=j1;k++){
            double t=x[i][k]+y[k][j];
            if (t>z[i][j]){z[i][j]=t;s[i][j]=k;}
         }
      }
}
```

square 计算 $z=x\otimes x$。

```
static void square(double [][]x,double [][]z)
{
   double [][]y=new double[n][n];
   for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)y[i][j]=x[i][j];
   qmult(x,y,z);
}
```

sum 计算 $x+y$。

```
static void sum(double [][]x,double [][]y)
{
   for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)x[i][j]+=y[i][j];
```

comp 计算 $\boldsymbol{D}+\boldsymbol{D}^4$。

```
static double comp()
{
   double [][]x=new double[n][n];
   double [][]y=new double[n][n];
   for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)x[i][j]=d[i][j];
   square(d,x);square(x,y);sum(d,y);
```

```
        double ans=0;
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)if(d[i][j]>ans)ans=d[i][j];
        return ans;
    }
```

实现算法的主函数如下：

```
    public static void main(String [] args)
    {
        input();
        output(comp());
    }
```

算法实现题 10-5　区间图最短路问题

★ 问题描述

S 是直线上 n 个带权区间的集合。从区间 $I\in S$ 到区间 $J\in S$ 的一条路是 S 的一个区间序列 $J(1),J(2),\cdots,J(k)$，其中 $J(1)=I,J(k)=J$，且对所有 $1\leqslant i\leqslant k-1$，$J(i)$ 与 $J(i+1)$ 相交。这条路的长度定义为路上各区间权之和。在所有从 I 到 J 的路中，路长最短的路称为从 I 到 J 的最短路。带权区间图的单源最短路问题要求计算从 S 中一个特定的源区间到 S 中所有其他区间之间的最短路。

★ 算法设计

对于给定 n 个带权区间，计算从最左区间到所有区间的最短路。最左区间是指 n 个区间中右端点最小的那个区间。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n，表示有 n 个带权区间。第 2 行起每行有 3 个正整数，分别表示带权区间的左端点、右端点和区间的权值。

★ 结果输出

将计算出的最左区间到所有区间的最短路之和输出到文件 output.txt 中。当最左区间与区间 i 不连通时，最左区间与区间 i 之间的最短路不计入。

输入文件示例	输出文件示例
input.txt	output.txt
10	500
1 4 15	
5 6 12	
3 7 13	
2 9 17	
10 13 17	
12 14 19	
11 15 21	
16 18 13	
17 19 15	
8 20 18	

分析与解答：

$O(n\alpha(n))$时间算法见主教材。

算法实现题 10-6 圆弧区间最短路问题

★ 问题描述

S 是给定圆上 n 个带权圆弧的集合。从圆弧 $I \in S$ 到圆弧 $J \in S$ 的一条路是 S 的一个圆弧序列 $J(1), J(2), \cdots, J(k)$，其中，$J(1)=I, J(k)=J$，且对所有 $1 \leqslant i \leqslant k-1$，$J(i)$与$J(i+1)$相交。这条路的长度定义为路上各圆弧权之和。在所有从 I 到 J 的路中，路长最短的路称为从 I 到 J 的最短路。带权圆弧图的单源最短路问题要求计算从 S 中一个特定的圆弧到 S 中所有圆弧之间的最短路。试设计解此问题的有效算法。

★ 算法设计

对于给定 n 个带权圆弧，计算从指定圆弧到所有圆弧的最短路。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行有 2 个正整数 n 和 m，表示有 n 个带权圆弧，并求第 m 个圆弧到所有圆弧的最短路。第 2 行起每行有 3 个整数，分别表示带权圆弧的左端点 a、右端点 b 和圆弧的权值 w。其中，左端点 a 和右端点 b 分别是按顺时针方向的圆心角，$0 \leqslant a < b \leqslant 21\,600$。

★ 结果输出

将计算出的第 m 个圆弧到所有圆弧的最短路之和输出到文件 output.txt 中。当圆弧 m 与圆弧 i 不连通时，圆弧 m 与圆弧 i 之间的最短路不计入。

输入文件示例	输出文件示例
input.txt	output.txt
4 1	160
0 10800 10	
5400 1620 50	
11100 18000 10	
17400 600 30	
160	

分析与解答：

圆弧区间最短路问题可以通过 2 次使用算法实现题 10-5 中的带权区间图最短路算法求解，算法见教材。

设指定的圆弧为$[a,b]$。在 a 处将圆切开，并拉成直线。以顺时针序排列各圆弧，将问题转化为区间图最短路问题。

接下来在 b 处将圆切开，并拉成直线。以逆时针序排列各圆弧，将问题转化为另一区间图最短路问题。

上述两个问题中，较短的最短路长圆弧区间最短路问题的解。

算法实现题 10-7 双机调度问题

★ 问题描述

F 大学计算机学院实验中心有两台相同的高性能超级计算机。学院高性能计算研究小

组的科学家们在进行一项复杂计算研究时，用分治策略将计算任务分解为 n 个互不相同的子任务 $J_1, J_2, \cdots, J_n$。每个子任务都需要 a 个时间单位来完成。由子任务间的逻辑关系给出执行这 n 个子任务间的 m 个先后次序。双机调度问题要求在两台相同的高性能超级计算机上确定 n 个子任务的最优调度方案，使全部完成 n 个子任务的时间最早。

★ **算法设计**

对于给定的 n 个互不相同的子任务 $J_1, J_2, \cdots, J_n$，以及这 n 个子任务间的 m 个先后次序，计算全部完成 n 个子任务的最早时间。假设超级计算机开始处理子任务的时间为0。

★ **数据输入**

由文件input.txt给出输入数据。第1行有2个正整数 n 和 m，表示有 n 个互不相同的子任务和 m 个子任务间的先后次序。接下来的 m 行中每行有2个正整数 x 和 y，表示子任务 x 应在子任务 y 之前完成。

★ **结果输出**

将计算出的最早完成时间输出到文件output.txt。在所有测试数据中，均假定完成每个子任务需要的时间单位为 $a=1$。

输入文件示例	输出文件示例
input.txt	output.txt
17 21	9
6 4	
7 4	
12 11	
11 8	
9 4	
9 5	
10 5	
4 1	
4 2	
5 3	
16 13	
8 5	
12 6	
12 7	
12 9	
16 4	
13 12	
14 12	
15 12	
7 5	
8 4	

分析与解答：

1）算法思想

给定的n个子任务间的m个先后次序可以表示为一个有向无环图DAG。从顶点x到顶点y有一条有向路时，称x是y的前驱，y是x的后继。特别地，当(x,y)是给定DAG中的一条边时，称x是y的直接前驱，y是x的直接后继。根据给定的DAG，可以将子任务按层序分类，如图10-4所示。子任务x的层序记为level(x)。

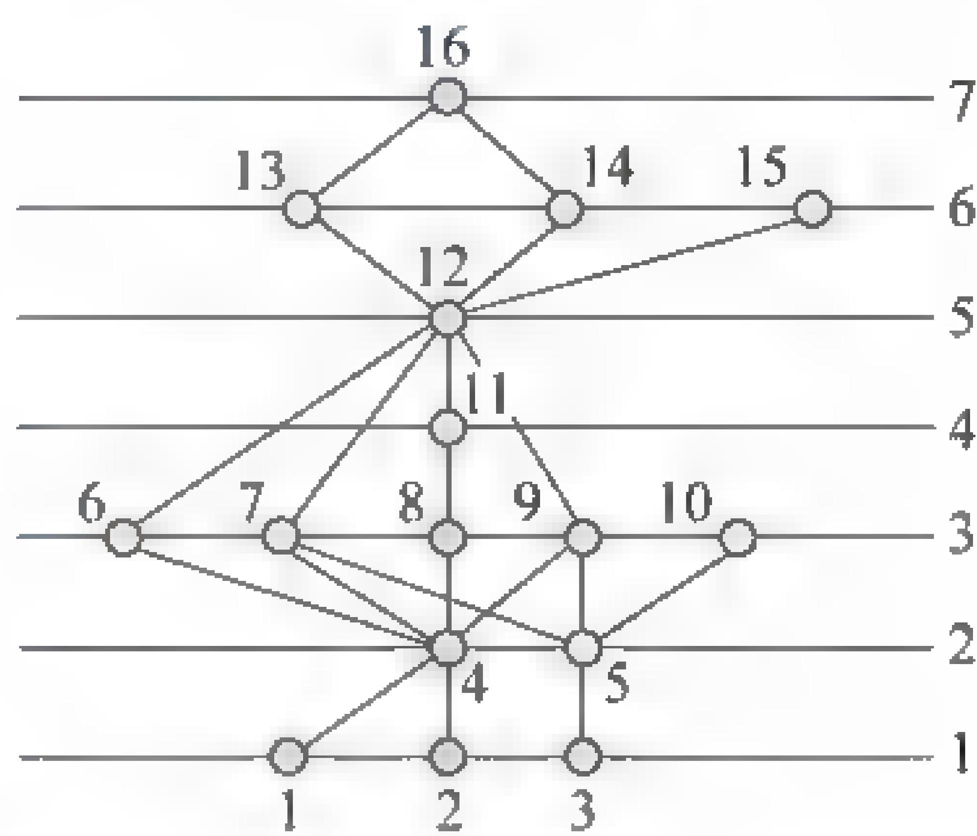


图10-4　有向无环图DAG的层序

设n个子任务分为L层。算法的基本思想是按照子任务的层序从高到低依次安排各子任务。假设在第$L,\cdots,i+1$层的子任务已安排，第i层还有u个子任务未安排，则可用$\lceil u/2\rceil$个单位时间安排这u个子任务如下：前$\lfloor u/2\rfloor$个单位时间里，每个时间可安排2个子任务。如果u是偶数，则完成第i层u个子任务的安排；当u是奇数时，第$\lceil u/2\rceil$个单位时间安排第i层的最后一个子任务和低于i层的另一子任务（也可不安排）。当u是奇数时，第i层称为一个奇层。当奇层i的最后一个单位时间安排了2个子任务x和y，则有level$(x)=i$，且level$(y)<i$。此时产生了一个从x到y的跳跃。

设所给定问题的奇层为$f_1>f_2>\cdots>f_k$。算法从第f_i层跳到第t_i层（当无法跳跃时$t_i=0$）。序列$(t_1,t_2,\cdots,t_k)$构成子任务调度的跳跃序列。算法的关键思想是，在有多个可选择的跳跃子任务时，选择按照字典序最大的跳跃序列。

2）算法描述

为了选择字典序最大的跳跃序列，采用2次扫描算法。

算法的第1次扫描对每个奇层猜测一个跳跃。在算法的第2次扫描时修正错误的猜测。

具体算法描述如下。

算法中用到的变量和数组说明如下：

```
static int  n,m,maxl=0;
```

n和m分别表示任务数和DAG的边数；maxl是DAG的层数。

```
static CircList []edges;       // DAG的边表
static CircList []levnode;     // DAG每层的顶点表
static CircList []rlist;       // 最高奇层表
static CircList []innode;      // 顶点入边表
static int []from;             // 跳跃的起跳顶点
static int []to;               // 跳跃的终跳顶点
static int []level;            // 顶点层序
static int []sub;              // 可替代终跳顶点
static int []t1;               // 与to对应的起跳顶点
static int []r;                // 最高可跳跃层
static int []freenode;         // 可选择的终跳顶点数
static int []flag;             // 标记数组
```

readin 读入初始数据，并初始化 DAG 的边表和顶点入边表。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    m=keyboard.readInt();
    edges=new CircList[n+1];
    innode=new CircList[n+1];
    level=new int[n+1];
    flag=new int[n+1];
    r=new int[n+1];
    t1=new int[n+1];
    for(int i=0;i<=n;i++){
        edges[i]=new CircList();
        innode[i]=new CircList();
        flag[i]=0;t1[i]=-1;r[i]=0;
    }
    for(int i=0;i<m;i++) {
        int x=keyboard.readInt();
        int y=keyboard.readInt();
        edges[x].add(0,new Integer(y));      // 插入边表
        innode[y].add(0,new Integer(x));     // 插入顶点入边表
    }
}
```

precomp 进一步计算层序和每层的顶点表。

```
static void precomp()
{
    for(int i=0;i<=n;i++)level[i]=-1;
    for(int i=1;i<=n;i++)
        if(innode[i].isEmpty()){
            Iterator it=edges[i].iterator();
            while (it.hasNext()){
                int x=((Integer)it.next()).intValue();
                int tmp=lev(x);
                if(level[i]<tmp)level[i]=tmp;
            }
            level[i]++;
        }
    for(int i=1;i<=n;i++)if(level[i]>maxl)maxl=level[i];
    levnode=new CircList[maxl+1];
    rlist=new CircList[maxl+1];
    sub=new int[maxl+1];
    from=new int[maxl+1];
    to=new int[maxl+1];
```

```
    freenode=new int[maxl+1];
    for(int i=0;i<=maxl;i++){
        sub[i]=0;from[i]=0;to[i]=0;freenode[i]=0;
        levnode[i]=new CircList();
        rlist[i]=new CircList();
    }
    for(int i=1;i<=n;i++)levnode[level[i]].add(0,new Integer(i));
}
```

lev 递归计算层序。

```
static int lev(int i)
{
    int x=0,maxl=0;
    if(level[i]>0) return level[i];
    else{
        level[i]=1;
        Iterator it=edges[i].iterator();
        while (it.hasNext()){
            x=((Integer)it.next()).intValue();
            int tmp=lev(x);
            if(maxl<tmp)maxl=tmp;
        }
        level[i]=maxl+1;
        return level[i];
    }
}
```

pass1 进行第 1 遍扫描,是算法的主体。

```
static void pass1()
{
    UnionFind lset=new UnionFind(maxl+1);
    for(int ti=maxl;ti>0;ti--){                    //按照层序从高到低进行计算
        //对当前层的每个顶点 y 计算 r[y]
        Iterator it=levnode[ti].iterator();
        while (it.hasNext()){
            int y=((Integer)it.next()).intValue();
            int rr=hlevel(y);
            if(rr<=maxl && to[rr]==0 && prefree(rr,y))r[y]=rr;
            else r[y]=lset.find(rr-1);
            rlist[r[y]].addlast(new Integer(y));
        }
        //计算最高跳跃层
        for(int f=maxl;f>ti;f    ){
            if(!rlist[f].isEmpty()){
                int y=((Integer)(rlist[f].remove (0))).intValue();
```

```
                    to[f]=y;t1[y]=f;
                    int g=lset.find(f-1);
                    lset.union(g,f);
                    rlist[g].concate(rlist[f]);
                }
            }
            int y=((Integer)rlist[ti].removelast()).intValue(); // 可替换顶点
            rlist[ti].clear();
            if(r[y]>ti)sub[ti]=y;
            if(!onelevel(ti)){to[ti]=-1;lset.union(ti-1,ti);}
            it=levnode[ti].iterator();
            while (it.hasNext()){
                y=((Integer)it.next()).intValue();
                if(free(y))freenode[ti]++;
            }
        }
    }
```

pass1 中的 hlevel 计算最高跳跃层。

```
static int hlevel(int y)
{
    int rr=maxl+1;
    Iterator it=innode[y].iterator();
    while (it.hasNext()){
        int x=((Integer)it.next()).intValue();
        int lev=t1[x];
        if(lev<0) lev=level[x];
        if(lev<rr)rr=lev;
    }
    return rr;
}
```

prefree 计算可自由选择的顶点。

```
static boolean prefree(int rr,int y)
{
    int cnt=0;
    Iterator it=innode[y].iterator();
    while (it.hasNext()){
        int x=((Integer)it.next()).intValue();
        if(level[x]==rr && free(x))cnt++;
    }
    return freenode[rr]>cnt;
}
```

free 判断顶点的自由性。

```
static boolean free(int x)
{
    return t1[x]<=r[sub[level[x]]];
}
```

onelevel 判断所在层是否为奇层。

```
static boolean onelevel(int ti)
{
    int cnt=0;
    Iterator it=levnode[ti].iterator();
    while (it.hasNext()){
        int y=((Integer)it.next()).intValue();
        if(t1[y]<0)cnt++;
    }
    return odd(cnt);
}
```

算法 pass2 进行第 2 遍扫描。

```
static void pass2()
{
    for(int f=1;f<=maxl;f++)
        if(to[f]>=0){
            if(to[f]>0)from[f]=afree(f);
            int g=mark(f);
            if(g>0)to[g]=sub[f];
        }
}
```

pass2 中的 afree 计算起跳顶点。

```
static int afree(int f)
{
    int y=-1;
    flg(f);
    Iterator it=levnode[f].iterator();
    while (it.hasNext()){
        int x=((Integer)it.next()).intValue();
        if(flag[x]==0 && free(x)){y=x;break;}
    }
    unflg(f);
    return y;
}
```

flg 用于标记直接前驱顶点。

```
static void flg(int f)
{
```

```
        Iterator it=innode[to[f]].iterator();
        while (it.hasNext()){
            int x=((Integer)it.next()).intValue();
            if(level[x]==f)flag[x]=1;
        }
}
```

unflg 用于撤除直接前驱顶点标记。

```
static void unflg(int f)
{
        Iterator it=innode[to[f]].iterator();
        while (it.hasNext()){
            int x=((Integer)it.next()).intValue();
            if(level[x]==f)flag[x]=0;
        }
}
```

mark 判定是否需替换顶点。

```
static int mark(int f)
{
        for(int i=1;i<=maxl;i++)
            if (t1[i]==from[f])return i;
        return 0;
}
```

compute 完成全部计算。

```
static int compute()
{
        precomp();
        pass1();
        pass2();
        int count=0;
        for(int i=1;i<=maxl;i++)if(to[i]==0)count++;
        count=(n+count+1)/2;
        return count;
}
```

实现算法的主函数如下：

```
public static void main(String [] args)
{
        readin();
        System.out.println(compute());
}
```

3) 算法复杂性

上述算法中,用到并查集的部分需要 $O(n\alpha(n))$时间。其余部分需要 $O(m+n)$时间。

因此，整个算法需要 $O(m+n\alpha(n))$ 时间和 $O(m+n)$ 空间。

算法实现题 10-8　离线最小值问题

★ 问题描述

给定集合 $S=\{1,2,\cdots,n\}$，以及由 n 个 Insert(x)和 m 个 DeleteMin()运算组成的运算序列。其中 n 个 Insert(x)运算将集合 $S=\{1,2,\cdots,n\}$ 中每个数插入动态集合 T 恰好一次，DeleteMin()每次删除动态集合 T 中的最小元素。离线最小值问题要求对于给定的运算序列，计算出每个 DeleteMin()运算输出的值。换句话说，要求计算数组 out，使第 i 次 DeleteMin()运算输出的值为 out[i]，$i=1,2,\cdots,m$。在执行具体计算前，运算序列已给定，这就是问题表述中离线的含义。

★ 算法设计

对于给定的由 n 个 Insert(x)和 m 个 DeleteMin()运算组成的运算序列，利用并查集编程计算出每个 DeleteMin()运算输出的值。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，分别表示运算序列由 n 个 Insert(x)和 m 个 DeleteMin()运算组成。第 2 行中有 $n+m$ 个整数。当整数 $x>0$ 时，则表示执行 Insert(x)运算；当整数 $x=-1$ 时，则表示执行 DeleteMin()运算。

★ 结果输出

将计算出的每个 DeleteMin()运算输出的值依次输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10 6	9 10 6 7 8 1
10 9 −1 −1 8 7 6 −1 −1 −1 5 4 3 2 1 −1	

分析与解答：

为了计算输出数组 out 的值，可以用一个优先队列 H，按照给定的运算序列依次执行 n 个 Insert(x)和 m 个 DeleteMin()运算，将第 i 次 DeleteMin()运算的结果记录到 out[i]中。执行完所给的运算后，数组 out 即为所求。在最坏情况下，这个算法需要 $O(m\log n)$ 计算时间。当 $m=\Omega(n)$ 时，算法需要的计算时间为 $O(n\log n)$。

实际上，上述算法是一个在线算法，即每次处理一个运算，并不要求事先知道运算序列。因而算法没有用到问题的离线性质。利用并查集和问题的离线性质可以将算法的计算时间进一步减少为 $O(n\alpha(n))$。

将给定的 n 个 Insert 和 m 个 DeleteMin 运算组成的运算序列表示为

$$I_1DI_2DI_3D\cdots I_kDI_{k+1}$$

其中，I_j（其中 $1\leqslant j\leqslant k+1$）为连续若干个（可以为 0）Insert 运算组成的运算序列，D 表示 DeleteMin 运算。下面用并查集算法模拟这个运算序列。开始时，将 I_j 中的 Insert 运算插入动态集合 T 中的元素用 union 运算组织成一个集合，并将该集合记为第 j 个集合，$1\leqslant j\leqslant k+1$。由于第 j 个集合的名与其序号可能不同，算法中用 2 个数组 si 和 is 来表示集合名与其序号的对应关系。例如，第 j 个集合名为 name 时，si[name]$=j$ 且 is[j]$=$name。另外，算法中还用到 2 个数组 prev 和 next 来表示 I_j 之间的顺序。开始时，prev[j]$=j-1$，$1\leqslant$

$j \leqslant k+1$ 且 $next[j]=j+1, 0 \leqslant j \leqslant k$。接下来,算法对每 i(其中 $1 \leqslant i \leqslant n$)用 find 运算计算出集合序号 j,使得 $i \in I_j$。这表明第 j 个 DeleteMin 运算输出元素 i,即 $out[j]=i$。然后用 union 运算将集合 I_j 与集合 I_{j+1} 合并,并修改数组 prev 和 next 的值,将 j 从链表中删除。算法结束后,输出数组 out 给出正确的计算结果。

```
static void offmin(int in[],int e[],int out[],int n,int k)
{
    int i,j;
    int []si;
    int []is;
    int []prev;
    int []next;
    FastUnionFind U=new FastUnionFind(n);
    si=new int[n+2];
    is=new int[n+2];
    prev=new int[k+2];
    next=new int[k+2];
    for(i=0;i<=n;i++){si[i]=0;is[i]=0;}
    for(i=0;i<=k;i++){prev[i+1]=i;next[i]=i+1;}
    prev[0]=0;
    for(i=1;i<=k;i++){
        int curr=(e[i]>e[i-1]) ? in[e[i-1]+1]:0;
        if(e[i]<i || e[i]<e[i-1]){System.out.println("Bad Input");return;}
        for(j=e[i-1]+2;j<=e[i];j++) curr=U.union(curr,in[j]);
        si[curr]=i;
        is[i]=curr;
    }
    for(i=1;i<=n;i++){
        int name=U.find(i);
        j=si[name];
        if(j<=k){
            int newset=name;
            if (is[next[j]]>0)newset=U.union(name,is[next[j]]);
            si[newset]=next[j];
            is[next[j]]=newset;
            next[prev[j]]=next[j];
            prev[next[j]]=prev[j];
            out[j]=i;
        }
    }
}
```

上面的算法中用两个数组 in 和 e 表示输入序列。in 给出 n 个元素的插入序列,e 给出 DeleteMin 运算在插入序列中的位置。例如,给定的插入元素和 DeleteMin 运算序列为 $\{3,4,D,2,D,1,D\}$ 时,有 $n=4$ 且 $k=3$。此时,in=[3,4,2,1]且 $e=[2,3,4]$;$I_1=[3,4]$,

$I_2=[2]$，$I_3=[1]$，$I_4=[\,]$。第1次执行算法主循环体时，$i=1$，此时找到 $j=3$，即 $1\in I_3$。由此可知 out[3]=1。算法将集合 I_3 与 I_4 合并后 $I_4=[1]$。当 $i=2$ 时，找到 $j=2$，即 $2\in I_2$。由此得 out[2]=2。算法将集合 I_2 与 I_4 合并后 $I_4=[1,2]$。同理当 $i=3$ 时，计算出 $j=1$。算法最后输出 out=[3,2,1]。

上述算法的主要计算量在于其主循环中的 n 个 find 运算。如果在执行 union 时总是将小树并到大树上，而且在执行 find 时，实行路径压缩，则 n 次 find 至多需要 $O(n\alpha(n))$ 时间。算法其余部分所需要的计算时间为 $O(n)$。由此可见，上述算法需要的总计算时间为 $O(n\alpha(n))$。

算法实现题 10-9　最近公共祖先问题

★ 问题描述

设计一个算法，对于给定的树中2个结点，返回它们的最近公共祖先。

★ 算法设计

对于给定的树 T，和树中结点对，计算结点对的最近公共祖先。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有1个正整数 n，表示给定的树有 n 个顶点，编号为 $1,2,\cdots,n$。编号为1的顶点是树根。接下来的 n 行中，第 $i+1$ 行描述与 i 个顶点相关联的子结点的信息。每行的第1个正整数 k 表示该顶点的儿子结点数；其后 k 个数中，每1个数表示1个儿子结点的编号。当 $k=0$ 时表示相应的结点是叶结点。

文件的第 $n+2$ 行是1个正整数 m，表示要计算最近公共祖先的 m 个结点对。接下来的 m 行，每行2个正整数，是要计算最近公共祖先的结点编号。结点编号可能重复出现。

★ 结果输出

将计算出的 m 个结点对的最近公共祖先结点编号输出到文件 output.txt。每行3个正整数，前2个是结点对编号，第3个是它们的最近公共祖先结点编号。

输入文件示例	输出文件示例
input.txt	output.txt
12	11 3 1
3 2 3 4	9 12 6
2 5 6	8 10 2
0	8 4 1
0	7 12 2
2 7 8	
2 9 10	
0	
0	
0	
2 11 12	
0	

```
0
5
3 11
7 12
4 8
9 12
8 10
```

分析与解答:

由于 m 个结点对已给定,可以用并查集离线计算结点对的最近公共祖先。具体算法是对树 T 作深度优先遍历。在遍历过程中,对于每个要计算最近公共祖先的结点对(v,w)作2次检查。第1次在结点 v 处检查,第2次在结点 w 处检查。在第2次检查时计算出(v,w)的最近公共祖先。具体算法描述如下。算法中用数组 anc 记录子树根结点编号。数组 mark 用于记录是否第2次检查结点对。

```
static void lca(int u)
{
    anc[U.find(u)]=u;
    Iterator it=edges[u].iterator();
    while (it.hasNext()){
        int v=((Integer)it.next()).intValue();
        lca(v);
        U.union (u,v);
        anc[U.find(u)]=u;
    }
    mark[u]=true;
    it=pnode[u].iterator();
    while (it.hasNext()){
        int v=((Integer)it.next()).intValue();
        if(mark[v]) out(u,v,anc[U.find (v)]);
    }
}
```

实现算法的主函数如下:

```
public static void main(String [] args)
{
    readin();
    lca(1);
}
```

readin 读入初始数据。

```
static void readin()
{
```

```
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    edges=new CircList[n+1];
    pnode=new CircList[n+1];
    anc=new int[n+1];
    mark=new boolean[n+1];
    for(int i=0;i<=n;i++){
        edges[i]=new CircList();
        pnode[i]=new CircList();
        mark[i]=false;
    }
    for(int i=1;i<=n;i++){
        int k=keyboard.readInt();
        for(int j=1;j<=k;j++){
            int x=keyboard.readInt();
            edges[i].add(0,new Integer(x));
        }
    }
    m=keyboard.readInt();
    for(int i=1;i<=m;i++){
        int x=keyboard.readInt();
        int y=keyboard.readInt();
        pnode[y].add(0,new Integer(x));
        pnode[x].add(0,new Integer(y));
    }
}
```

上述算法显然需要$O((m+n)\alpha(m+n))$时间和$O(m+n)$空间。

算法实现题 10-10　达尔文芯片问题

★ 问题描述

人的大脑里发生的一切是神奇的，甚至是不可理解的，正是这种神奇使得人具有自我意识。如果用普通硅片、电路、传感器制成的机器人也能进化，从而能有意识的行动，那么是否有一天，机器人也会变得和人一样有意识？计算机的硬件也许能像自然界人类和其他生物进化的方式进行进化这一想法，早在20世纪60年代就被提出，但如何着手是到1998年因美籍华裔计算机科学家的一个灵感才得以突破。这一灵感就是被称为达尔文芯片的高集成度可编程集成电路块，简称DPGA。

最近，F大学计算机学院计算机神经学研究小组的科学家们发现，对达尔文芯片的关键逻辑元进行重组后产生F一种奇特的现象。将若干关键逻辑元按照电路板平面坐标系二维降序排列，经过电路演化，这些关键逻辑元自动按照x坐标和y坐标方向延伸连接成一棵树。这棵树的每条边都平行于x坐标轴或平行于y坐标轴。关键逻辑元构成这棵树的全部叶结点。这类树称为以关键逻辑元为叶结点的正交树。有趣的是，达尔文芯片自动产生的正交树的总边长是所有这种正交树中总边长最小的。例如，将5个关键逻辑元分别置

于电路板 xOy 坐标系中(1,5),(2,4),(3,3),(4,2)和(5,1)处,则达尔文芯片自动产生的一棵正交树如图 10-5 所示,它的总边长为 12。

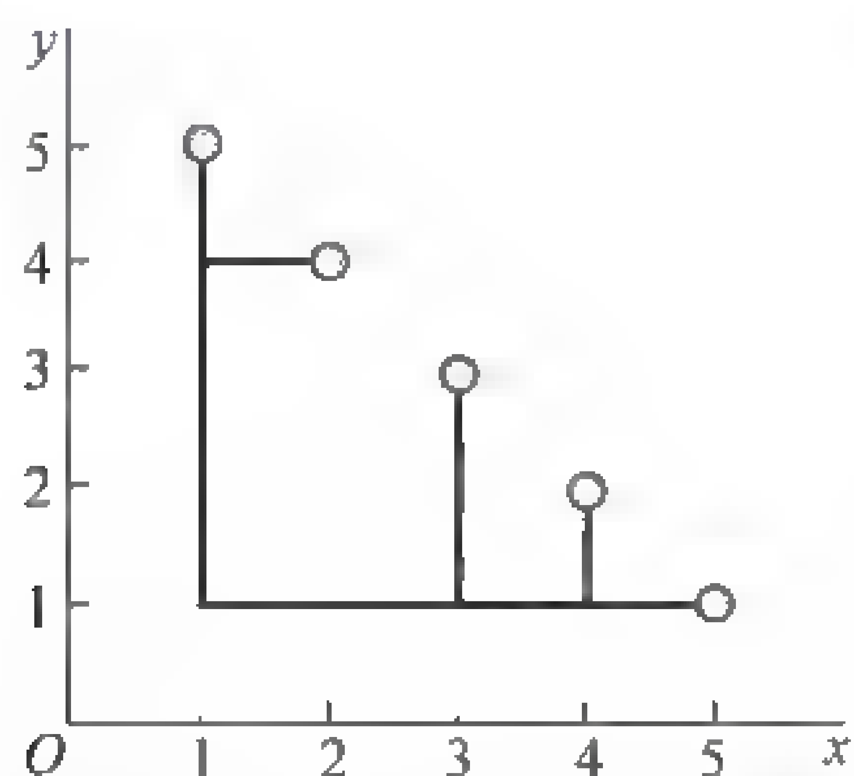


图 10-5 达尔文芯片正交树

★ 算法设计

给定电路板 xOy 坐标系,以及电路板上 n 个关键逻辑元在 xOy 坐标系中按照二维降序排列的位置 (x_1, y_1), (x_2, y_2), …, (x_n, y_n)。其中, $1 \leqslant n \leqslant 3000$, $0 \leqslant x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n \leqslant 20\,000$, $0 \leqslant y_n \leqslant y_{n-1} \leqslant \cdots \leqslant y_1 \leqslant 20\,000$。

计算以 n 个关键逻辑元为叶结点的正交树中,总边长最小的最优正交树。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行中的整数为关键逻辑元个数 n。接下来的 n 行中每行 1 个整数,依次为 $x_1, x_2, \cdots, x_n$。最后的 n 行中每行一个整数,依次为 $y_1, y_2, \cdots, y_n$。

★ 结果输出

将所找到的最优正交树总边长的值,输出到文件 output.txt。精确到小数点后 2 位。

输入文件示例	输出文件示例
input.txt	output.txt
5	12
1	
2	
3	
4	
5	
5	
4	
3	
2	
1	

分析与解答:

设叶结点为 (x_i, y_i), (x_{i+1}, y_{i+1}), …, (x_j, y_j) 的最优正交树总边长的值为 $m(i,j)$,则动态规划递归式为

$$m(i,j) = \min_{i \leqslant s < j} \{m(i,s) + m(s+1,j) + x_{s+1} - x_i + y_s - y_j\}$$

按此式设计的动态规划算法需要 $O(n^3)$ 计算时间。进一步分析表明,函数 $h(i,s,j) = x_{s+1} - x_i + y_s - y_j$ 满足四边形不等式。从而可将计算时间减至 $O(n^2)$。

具体算法描述如下:

```
static void SpeedDynamic(double [ ]x,double [ ]y,int n,double [ ][ ]m,int [ ][ ]s)
{
    for (int i=1; i<=n; i++) {m[i][i]=0;s[i][i]=i;}
```

```
for (int r=2; r<=n; r++)
    for (int i=1; i<=n-r+1; i++) {
        int j=i+r-1;
        int i1=s[i][j-1];
        int j1=s[i+1][j]<j?s[i+1][j]:j-1;
        m[i][j]=m[i][i1]+m[i1+1][j]+x[i1+1]+y[i1]-x[i]-y[j];
        s[i][j]=i1;
        for (int k=i1+1; k<=j1; k++) {
            double t=m[i][k]+m[k+1][j]+x[k+1]+y[k]-x[i]-y[j];
            if (t<m[i][j]) {
            m[i][j]=t;
            s[i][j]=k;}
        }
    }
}
```

算法实现题 10-11　多柱 Hanoi 塔问题

★ 问题描述

多柱 Hanoi 塔问题是 3 柱 Hanoi 塔问题的推广。在一般情况下，给定 p 个塔座 1，2，…，p。开始时，在塔座 1 上有一叠共 n 个圆盘，这些圆盘自下而上、由大到小地叠在一起。各圆盘从小到大编号为 1，2，…，n。现要求将塔座 1 上的这一叠圆盘移到塔座 2 上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则(1)：每次只能移动 1 个圆盘。

规则(2)：任何时刻都不允许将较大的圆盘压在较小的圆盘之上。

规则(3)：在满足移动规则(1)和(2)的前提下，可将圆盘移至 1，2，…，p 中任一塔座。

设计一个算法，计算完成所要求移动的最少移动次数。

★ 算法设计

对于给定的 p 个塔座和塔座 1 上的 n 个圆盘，计算将 n 个圆盘移到塔座 2 上需要的最少移动次数。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行中的 2 个正整数 n 和 p 分别表示有 n 个圆盘和 p 个塔座。

★ 结果输出

将计算出的最少移动次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 4	13

分析与解答：

设 $m(n,p)$ 是解 n 个圆盘，p 个柱的 Hanoi 塔问题所需的最少移动次数，则 $m(n,p)$ 满足如下的动态规划递归式

$$m(n,p)=\begin{cases}1 & n=1,\ p\geqslant 2\\ \min\limits_{0\leqslant k\leqslant n-1}\{2m(k,p)+m(n-k,p-1)\} & n>1,\ p>2\end{cases}$$

由上述递归式计算出最优分割点 k 后，最优移动序列分3步完成如下：

(1) 将源柱1上的 k 个圆盘递归地移动到任意空闲柱 $r(1<r<p)$ 上。需要 $m(k,p)$ 次移动。

(2) 将源柱1上剩余的 $n-k$ 个圆盘递归地移动到目的柱 p 上，且在移动过程中不使用柱 r。这需要 $m(n-k,p-1)$ 次移动。

(3) 将柱 r 上的 k 个圆盘递归地移动到目的柱 p 上。这需要 $m(k,p)$ 次移动。

设 $F(n,p,k)=2m(k,p)+m(n-k,p-1),n>1,p>2,0\leqslant k\leqslant n-1$。

集合 $\Pi(n,p)=\{k\in Z\mid F(n,p,k)=m(n,p)\}$ 称为 $m(n,p)$ 的最优分割点集合。

设 $\min k(n,p)=\min\limits_{k\in\Pi(n,p)}\{k\}$，$\max k(n,p)=\max\limits_{k\in\Pi(n,p)}\{k\}$。

易知，当 $p\geqslant 2,n\leqslant 1$ 时，$\min k(n,p)=\max k(n,p)=0$。当 $p=3$ 时，对任意 $n\geqslant 1$ 有，$\min k(n,3)=\max k(n,3)=n-1$。更进一步，对于任意 $p\geqslant 3,n\geqslant 1$，有

(1) $\Pi(n,p)=\{k\in Z\mid \min k(n,p)\leqslant k\leqslant \max k(n,p)\}$；

(2) $\min k(n,p)\leqslant \min k(n+1,p)\leqslant \min k(n,p)+1$，

$\max k(n,p)\leqslant \max k(n+1,p)\leqslant \max k(n,p)+1$。

为了便于叙述，定义函数 $h(x,p)$ 为

$$h(x,p)=\binom{p+x-3}{p-2}$$

利用组合恒等式 $\binom{r}{k}=\binom{r-1}{k}+\binom{r-1}{k-1}$ 可以推知，$h(x,p)=h(x-1,p)+h(x,p-1)$。

由于 $h(x,p)$ 是严格单调递增函数，其逆函数 $f(x,p)$ 可以定义为

$$f(x,p)=h^{-1}(x,p)=\max\{k\mid h(k,p)\leqslant x\}$$

对于任意 $p\geqslant 3,s\geqslant 1$，当 $n=h(s,p)$ 时，有

$\min k(n,p)=\max k(n,p)=h(s-1,p)$，即此时有 $\Pi(h(s,p),p)=\{h(s-1,p)\}$。

由此可知，对于任意 $p\geqslant 3,n\geqslant 1$，设 $s=f(n,p)=\max\{k\mid h(k,p)\leqslant n\}$，则有

(1) $\min k(n,p)=\begin{cases}h(s-1,p) & h(s,p)\leqslant n\leqslant h(s,p)+h(s+1,p-2)\\ n-h(s+1,p-1) & h(s,p)+h(s+1,p-2)\leqslant n\leqslant h(s+1,p)\end{cases}$。

(2) $\max k(n,p)=\begin{cases}n-h(s,p-1) & h(s,p)\leqslant n\leqslant h(s,p)+h(s,p-1)\\ h(s,p) & h(s,p)+h(s,p-1)\leqslant n\leqslant h(s+1,p)\end{cases}$。

进一步可得，对于任意 $p\geqslant 3,n\geqslant 1$，设 $s=f(n,p)=\max\{k\mid h(k,p)\leqslant n\}$。取 $k(n,p)=h(s-1,p)+s(n-h(s,p))/(p+s-3)$，则有 $k(n,p)\in\Pi(n,p)$。

基于以上的讨论，可以将一般情况下的多柱 Hanoi 塔问题的动态规划算法转化为一个简单的递归算法如下：

$$m(n,p)=\begin{cases}1 & n=1,\ p\geqslant 2\\ 2m(k(n,p),p)+m(n-k(n,p),p-1) & n>1,\ p>2\end{cases}$$

```
static int m(int n,int p)
{
```

```
    int s=f(n,p);
    int sum=1,j=1;
    for (int t=1;t<s;t++){
        j*=2;
        sum+=j*h(t+1,p-1);
    }
    sum+=2*j*(n-h(s,p));
    return sum;
}
```

算法中的函数 $h(s,p)$用于计算$\binom{p+s-3}{p-2}$；$f(n,p)$用于计算 $\max\{k \mid h(k,p)\leqslant n\}$。

```
static int f(int n,int p)
{
    int i;
    for (i=1;h(i,p)<n;i++);
    return (h(i,p)>n)?i-1:i;
}
static int h(int s,int p)
{
    return c[p+s-3][p-2];
}
static void computec()
{
    int r=split(n,p-2)+p-2;
    c=new int[r+1][p+1];
    comb(r,p-2);
}
static void comb(int n,int m)
{
    for (int i=0;i<=n;i++)c[i][0]=1;
    for (int j=1;j<=m;j++)c[0][j]=0;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            c[i][j]=c[i-1][j-1]+c[i-1][j];
}
static int split(int n,int m)
{
    int i,last=0;
    int []a=new int[m+1];
    for (int j=1;j<=m;j++)a[j]=0;
    for (i=1;(i<m)||(last<=n);i++){
        for (int j=m;j>1;j--)a[j]+=a[j-1];
        a[1]++;
```

```
            if (i>=m) last=a[m];
        }
        return i-m-1;
}
```

算法实现题 10-12　线性时间 Huffman 算法

★ 问题描述

在一个操场的四周摆放着 n 堆石子。每堆的石子数不超过 $10\times n$。现要将石子有次序地合并成一堆。规定每次只能选 2 堆石子合并成新的一堆,合并的费用为新的一堆的石子数。试设计一个线性时间算法,计算出将 n 堆石子合并成一堆的最小总费用。

★ 算法设计

对于给定 n 堆石子,计算合并成一堆的最小总费用。

★ 数据输入

由文件 input.txt 提供输入数据。文件的第 1 行是正整数 n,表示有 n 堆石子。第 2 行有 n 个数,分别表示每堆石子的个数。

★ 结果输出

将计算出的最小总费用输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
6	224
45 13 12 16 9 5	

分析与解答:

设 n 堆石子的个数分别为 $a[i], i=1,2,\cdots,n$。可以用标准 Huffman 算法求解,需要 $O(n\log n)$时间。

注意到 $a[i]\leqslant 10*n$,即 $a[i]=O(n)$。可以进一步将所需时间减少至 $O(n)$。

(1) 先用 $O(n)$时间将 $a[i]$排序。

(2) 注意到 Huffman 算法产生的所有内结点是从小到大排列的,可以用一个队列存放算法产生的内结点。求当前合并子树时,只要比较最小内结点和最小外结点即可。

按照上述思想可以将 Huffman 算法的时间减少至 $O(n)$。

```
public static void linearHF()
{
    int i,j,k,x,y;
    sum=0;
    BinSort.sort(L,n);                  // L 为输入序列,线性时间排序
    for(i=0;i<n;i++)a[n+i]=L[i];        // a[n]~a[2*n-1]为排好序的外结点
    a[2*n]=Integer.MAX_VALUE ;
    i=n+1;j=n-1;k=n;x=n;
    while(true){
        // y 为右指针
        if(j<k || a[i]<=a[j])y=i++;
        else y=j     ;
```

```
        L[k]=x;                    // 左儿子结点
        R[k]=y;                    // 右儿子结点
        a[k]=a[x]+a[y];            // 新内结点,a[k]~a[j]为内结点队列
        sum+=a[k];
        if(k==1)break;
        // x为左指针
        if(a[i]<=a[j])x=i++;
        else x=j--;
    }
}
```

算法实现题 10-13 单机调度问题

★ **问题描述**

F大学计算机学院实验中心有一台高性能超级计算机。学院高性能计算研究小组的科学家们在进行一项复杂计算研究时,用分治策略将计算任务分解为 n 个互不相同的子任务 $J_1,J_2,\cdots,J_n$。第 i 个子任务需要时间 t_i 和空间 s_i。超级计算机将 n 个子任务依次划分成若干连续时间段进行计算。新时间段开始工作之前需要系统调整时间 x。假设第 k 个时间段中的子任务是 $J_p,J_{p+1},\cdots,J_q$;第 k 个时间段的结束时间为 f_k,则第 k 个时间段的费用定义为 $c_k=\sum_{i=p}^{q}s_if_k$。完成 n 个子任务的总费用为各时间段费用之和。单机调度问题要求确定 n 个子任务的最优时间段划分,使全部完成 n 个子任务的总费用最小。

★ **算法设计**

对于给定的 n 个互不相同的子任务 $J_1,J_2,\cdots,J_n$ 和这 n 个子任务需要时间和空间,以及系统调整时间 x,计算全部完成 n 个子任务的最小费用。

★ **数据输入**

由文件 input.txt 给出输入数据。第1行有2个整数 n 和 x,表示有 n 个互不相同的子任务,新时间段开始工作之前需要系统调整时间 x。接下来的 n 行中每行有2个整数 t 和 s,表示相应的子任务需要时间 t 和空间 s。

★ **结果输出**

将计算出的最小费用输出到文件 output.txt。在所有测试数据中,均假定从时间0开始。

输入文件示例	输出文件示例
input.txt	output.txt
5 1	153
1 3	
3 2	
4 3	
2 3	
1 4	

分析与解答:

1) 动态规划算法

设完成子任务 $J_1,J_2,\cdots,J_i$ 的最小费用为 $d[i]$,$i=1,2,\cdots,n$。计算 $d[i]$ 的动态规划递

归式如下。$d[j]=\min\limits_{1\leqslant i<j}\{d[i]+w(i,j)\}$。其中，$w(i,j)$是将 $J_i,J_{i+1},\cdots,J_j$ 划分为一个新时间段的费用。

据此可设计动态规划算法如下：

```
static void dyna()
{
    for(int j=1;j<=n;j++){
        for(int i=0;i<j;i++){
            int tmp=d[i]+w(i,j);
            if(d[j]>tmp)d[j]=tmp;
        }
    }
}
```

$w(i,j)$计算将 $J_i,J_{i+1},\cdots,J_j$ 划分为一个新时间段的费用。

```
static int w(int i,int j)
{
    if(i>n || j>n)return Integer.MAX_VALUE;
    return (s+t[j]-t[i]) * (f[n]-f[i]);
}
```

其中，s 为系统调整时间，t 和 f 根据输入数据初始化。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    s=keyboard.readInt();
    t=new int [n+1];
    f=new int [n+1];
    d=new int [n+1];
    t[0]=f[0]=d[0]=0;
    for (int i=0; i<n; i++){
        int x=keyboard.readInt();
        int y=keyboard.readInt();
        t[i+1]=t[i]+x;f[i+1]=f[i]+y;
        d[i+1]=Integer.MAX_VALUE;
    }
}
```

上述算法显然需要 $O(n^2)$计算时间。

2) 算法优化

对于形如 $d[j]=\min\limits_{1\leqslant i<j}\{d[i]+w(i,j)\}$的动态规划递归式，当函数 $w(i,j)$满足四边形不等式时，可将计算时间从 $O(n^2)$减至 $O(n\log n)$。

容易证明，本题中函数 $w(i,j)$满足四边形不等式

$$w(i,j)+w(i',j')\leqslant w(i',j)+w(i,j'),\quad i\leqslant i'<j\leqslant j'$$

改进的算法依次计算 $d[i]$，每次用 $O(\log n)$ 时间，因此总共耗时 $O(n\log n)$。

设 $c(i,j)=d[i]+w(i,j)$，且 $w(i,j)$ 满足四边形不等式。

改进算法的主要依据是：

① 对于给定的 $k<l\leqslant n$，$f(r)=c(l,r)-c(k,r)$ 是单调非减函数。

② 对于给定的 $k<l<j\leqslant n$，若 $c(l,j)\leqslant c(k,j)$，则对所有 $j\leqslant t\leqslant n$ 有 $c(l,t)\leqslant c(k,t)$。

③ 对于给定的 $k<l<j\leqslant n$，若 $c(l,j)>c(k,j)$，

设 $h(l,k)=\min\limits_{l<i\leqslant n}\{i \mid c(l,i)\leqslant c(k,i)\}$，则对所有 $l<t<h$，有 $c(l,t)>c(k,t)$；对所有 $h\leqslant t\leqslant n$，有 $c(l,t)\leqslant c(k,t)$。

④ 对于给定的 $k<l<n$，$h(l,k)\leqslant t$，当且仅当 $c(l,t)\leqslant c(k,t)$。

⑤ 对于给定的 $k<l\leqslant n$，可用二分搜索算法在 $O(\log n)$ 时间内计算 $h(l,k)$。

算法用一个队列来存储需要搜索的下标下界 k 和上界 h。根据性质①～⑤适时修改下标搜索空间。具体算法描述如下。

```
static void speedup()
{
    ArrayQueue Q=new ArrayQueue(n+1);
    pNode p=new pNode(0,1);
    Q.put(p);
    for(int j=1;j<=n;j++){
        p=(pNode)Q.getFrontElement();
        int l=p.k;
        int c1=c(j-1,j);
        int c2=c(l,j);
        if(c1<=c2){
            d[j]=c1;
            Q.clear();
            p=new pNode(j-1,j+1);
            Q.put(p);
        }
        else{
            d[j]=c2;
            while(true){
                p=(pNode)Q.getRearElement();
                if(p!=null && c(j-1,p.h )<=c(p.k,p.h))Q.removeRear();
                else break;
            }
            p=(pNode)Q.getRearElement();
            int kk=n+1;
            if(p!=null)kk=p.k;
            int h=search(j-1,kk);
            p=new pNode(j-1,h);
            Q.put(p);
            p=(pNode)Q.getElement(1);
            if(p!=null && j+1==p.h)Q.remove();
```

```
                    else {
                        p=(pNode)Q.getFrontElement();
                        p=new pNode(p.k,p.h+1);
                        Q.change(p);
                    }
                };
            }
        }
```

其中,$c(i,j)$用$O(1)$时间计算$c(i,j)=d[i]+w(i,j)$;search(l,k)用二分搜索算法计算$h(l,k)$,需要的计算时间为$O(\log n)$。因此,整个算法所需的计算时间为$O(n\log n)$。

算法实现题 10-14　最大费用单机调度问题

★ 问题描述

F大学计算机学院实验中心有一台高性能超级计算机。学院高性能计算研究小组的科学家们在进行一项复杂计算研究时,用分治策略将计算任务分解为n个互不相同的子任务$J_1,J_2,\cdots,J_n$。第i个子任务需要时间t_i和空间s_i。超级计算机将n个子任务依次划分成若干连续时间段进行计算。新时间段开始工作之前需要系统调整时间x。假设第k个时间段中的子任务是$J_p,J_{p+1},\cdots,J_q$;第k个时间段的结束时间为f_k,则第k个时间段的费用定义为$c_k=\sum_{i=p}^{q}s_i f_k$。完成$n$个子任务的总费用为各时间段费用之和。最大费用单机调度问题要求确定n个子任务的最优时间段划分,使全部完成n个子任务的总费用最大。

★ 算法设计

对于给定的n个互不相同的子任务$J_1,J_2,\cdots,J_n$和这n个子任务需要时间和空间,以及系统调整时间x,计算全部完成n个子任务的最大费用。

★ 数据输入

由文件input.txt给出输入数据。第1行有2个整数n和x,表示有n个互不相同的子任务,新时间段开始工作之前需要系统调整时间x。接下来的n行中每行有2个整数t和s,表示相应的子任务需要时间t和空间s。

★ 结果输出

将计算出的最大费用输出到文件output.txt。在所有测试数据中,均假定从时间0开始。

输入文件示例	输出文件示例
input.txt	output.txt
5 1	180
1 3	
3 2	
4 3	
2 3	
1 4	

分析与解答:

1) 动态规划算法

设完成子任务$J_1,J_2,\cdots,J_i$的最大费用为$d[i]$,$i=1,\cdots,n$。计算$d[i]$的动态规划递

归式为 $d[j]=\max\limits_{1\leqslant i<j}\{d[i]+w(i,j)\}$。其中，$w(i,j)$ 是将 $J_i,J_{i+1},\cdots,J_j$ 划分为一个新时间段的费用。

据此可设计动态规划算法如下：

```
static void dyna()
{
    for(int j=1;j<=n;j++){
        for(int i=0;i<j;i++){
            int tmp=d[i]+w(i,j);
            if(d[j]<tmp)d[j]=tmp;
        }
    }
}
```

$w(i,j)$ 计算将 $J_i,J_{i+1},\cdots,J_j$ 划分为一个新时间段的费用。

```
static int w(int i,int j)
{
    return (s+t[j]-t[i]) * (f[n]-f[i]);
}
```

其中，s 为系统调整时间，t 和 f 根据输入数据初始化。

```
static void readin()
{
    ReadStream keyboard=new ReadStream();
    n=keyboard.readInt();
    s=keyboard.readInt();
    t=new int [n+1];
    f=new int [n+1];
    d=new int [n+1];
    t[0]=f[0]=d[0]=0;
    for (int i=0; i<n; i++){
        int x=keyboard.readInt();
        int y=keyboard.readInt();
        t[i+1]=t[i]+x;f[i+1]=f[i]+y;
        d[i+1]=0;
    }
}
```

上述算法显然需要 $O(n^2)$ 计算时间。

2）算法优化

将动态规划递归式 $d[j]=\max\limits_{1\leqslant i<j}\{d[i]+w(i,j)\}$ 变换为等价的递归式

$$d[j]=\min_{1\leqslant i<j}\{d[i]-w(i,j)\}$$

对于形如 $d[j]=\min\limits_{1\leqslant i<j}\{d[i]+w(i,j)\}$ 的动态规划递归式，当函数 $w(i,j)$ 满足四边形不等式时，可将计算时间从 $O(n^2)$ 减至 $O(n\log n)$。

容易证明,本题中函数 $w(i,j)$满足四边形不等式,从而 $-w(i,j)$满足反四边形不等式。

当函数 $w(i,j)$满足反四边形不等式时,也可将计算时间从 $O(n^2)$减至 $O(n\log n)$。

改进的算法依次计算 $d[i]$,每次用 $O(\log n)$时间,因此总共耗时 $O(n\log n)$。

设 $c(i,j)=d[i]+w(i,j)$,且 $w(i,j)$满足反四边形不等式。

改进算法的主要依据如下:

① 对于给定的 $k<l\leqslant n$,$f(r)=c(l,r)-c(k,r)$是单调非增函数。

② 对于给定的 $k<l<j\leqslant n$,若 $c(l,j)\leqslant c(k,j)$,则对所有 $j\leqslant t\leqslant n$ 有 $c(l,t)\leqslant c(k,t)$。

③ 对于给定的 $k<l<j\leqslant n$,若 $c(l,j)>c(k,j)$,设 $h(l,k)=\min\limits_{k<i\leqslant n}\{i\,|\,c(l,i)\leqslant c(k,i)\}$,则对所有 $k<t<h$,有 $c(l,t)>c(k,t)$;对所有 $h\leqslant t\leqslant n$,有 $c(l,t)\leqslant c(k,t)$。

④ 对于给定的 $l<k<n$,$h(l,k)\leqslant t$,当且仅当 $c(l,t)\leqslant c(k,t)$。

⑤ 对于给定的 $l<k\leqslant n$,可用二分搜索算法在 $O(\log n)$时间内计算 $h(l,k)$。

改进算法用一个栈来存储需要搜索的下标下界 k 和上界 h。根据性质①~⑤适时修改下标搜索空间。具体算法描述如下:

```
static void speedup()
{
    ArrayStack S=new ArrayStack(n+1);
    pNode p=new pNode(0,n+1);
    S.push(p);
    for(int j=1;j<=n;j++){
        p=(pNode)S.peek();
        int l=p.k;
        int c1=c(j-1,j);
        int c2=c(l,j);
        if(c1>c2)d[j]=c2;
        else{
            d[j]=c1;
            while(!S.empty()){
                p=(pNode)S.peek();
                if(c(j-1,p.h-1)<c(p.k,p.h-1))S.pop();
                else break;
            }
            if(S.empty())S.push(new pNode(j-1,n+1));
            else{
                p=(pNode)S.peek();
                int h=search(p.k,j-1);
                S.push (new pNode(j-1,h));
            }
        }
        p=(pNode)S.peek();
        if(p.h==j+1)S.pop();
    }
}
```

其中，$c(i,j)$用$O(1)$时间计算$c(i,j)=d[i]+w(i,j)$；search(l,k)用二分搜索算法计算$h(l,k)$，需要的计算时间为$O(\log n)$。因此，整个算法所需的计算时间为$O(n\log n)$。

算法实现题 10-15　飞机加油问题

★ 问题描述

F国际航空公司在世界范围有n个国际机场。第i个国际机场到中心机场的距离为d_i，$i=1,\cdots,n$。从国际机场j到国际机场i的飞行费用为$w(i,j)=s+(d_j-d_i)^2$，s为地面加油费用。从任何国际机场飞往中心机场的飞机可以在任一国际机场加油后继续飞行。飞机加油问题要求确定从距中心机场最远的国际机场飞到中心机场的最少费用。

★ 算法设计

对于给定的n个国际机场到中心机场的距离$d_1,d_2,\cdots,d_n$，以及地面加油费用s，计算从距中心机场最远的国际机场飞到中心机场的最少费用。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有2个整数n和s，表示有n个国际机场(不包括中心机场)，地面加油费用s。第2行中每行有n个整数$d_1,d_2,\cdots,d_n$，表示给定的n个国际机场到中心机场的距离。

★ 结果输出

将计算出的最小费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	64
1 3 6 7 10	

分析与解答：

先将国际机场按照到中心机场的距离从小到大排序。设从国际机场j到中心机场的最小费用为$c[j]$，$j=1,2,\cdots,n$。计算$c[j]$的动态规划递归式为$c[j]=\max\limits_{1\le i<j}\{c[i]+w(i,j)\}$。其中，$w(i,j)$是从国际机场$j$到国际机场$i$的飞行费用。

其余与算法实现题 10-15 相同。

第11章 在线算法设计

习题 11-1　在线算法 LFU 的竞争性

证明对任何非负常数 α,在线算法 LFU 都不是 α 竞争的。

分析与解答:

设 $p_1,p_2,\cdots,p_k,p_{k+1}$ 是内存中 $k+1$ 个不同的页面。考查内存访问请求序列 $\sigma=\sigma_1+\sigma_2$,其中 $\sigma_1=p_1,p_2,p_2,\cdots,p_k,p_k$,即访问页面 p_1 的次数为 1,访问页面 p_i 的次数为 2,$2\leqslant i\leqslant k$。$\sigma_2=p_{k+1},p_1,p_{k+1},p_1\cdots$。

对于内存访问请求序列 σ,在 σ_1 以后算法 LFU 的耗费为 σ_2 ,而最优离线算法的耗费为 1。由此可见,对任何非负常数 α,在线算法 LFU 都不是 α 竞争的。

习题 11-2　多读写头磁盘问题的在线算法

磁盘上的磁道是按照同心圆划分的。在一个多读写头磁盘系统中有 k 个磁头读取磁盘上存储的数据。当系统接收到一个数据访问请求时,系统要在线确定由哪一个磁头来读取数据。试设计一个完成上述任务的在线算法,并分析算法的竞争比。

分析与解答:

由于 k 个磁头在直线上移动,k 读写头磁盘问题实际上是直线上的 k 服务问题。用主教材中关于直线上 k 服务问题的对称移动算法可以直接得到 k 读写头磁盘问题的竞争比为 k 的在线算法。

习题 11-3　带权页调度问题

在带权页调度问题中,高速缓存中的 k 个页面编号为 $1,2,\cdots,k$,将低速内存中的一个页面调入高速缓存 i 的费用为 w_i。试设计带权页调度问题的在线算法,并分析算法的竞争比。

分析与解答:

带权页调度问题实际上是 k 服务问题的特殊情形。用主教材中关于 k 服务问题的平衡算法可以直接得到带权页调度问题的竞争比为 k 的在线算法。

算法实现题 11-1　最优页调度问题

★ 问题描述

页调度问题是系统软件设计中提出的一个基本问题。系统软件在进行内存管理时,将内存按其存取速度分成 2 级,即高速缓存和低速内存。内存被分成固定大小的页面进行管

理。高速缓存可容纳 k 个页面，其他页面在低速内存中。页调度问题的输入是内存访问请求序列 $\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$。当内存访问请求要访问的页面 $\sigma(i)$ 在高速缓存中时，不需页面调度；而当页面 $\sigma(i)$ 不在高速缓存中时，发生页面缺失，调度算法要确定高速缓存中与 $\sigma(i)$ 交换的页面。页调度算法对于内存访问请求序列 $\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$ 的耗费是算法在执行过程中产生的页面缺失次数。内存访问请求是无法预知的，它是随着时间的推移一个接着一个地给出的。最优页调度问题要求响应每一个内存访问请求，并且使发生页面缺失的总次数最少。

★ 算法设计

对于给定的 2 级内存页面分布以及内存访问请求序列 $\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$，设计一个算法，给出响应每一个内存访问请求的最优页面调度，使发生页面缺失的总次数最少。

★ 数据输入

由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n,k 和 m，表示有 n 个内存页面 $1,2,\cdots,n$。高速缓存可容纳 k 个页面。初始时页面 $1,2,\cdots,k$ 在高速缓存中。内存访问请求序列的长度为 m。第 2 行有 m 个正整数表示内存访问请求序列 $\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$，$1\leqslant\sigma(i)\leqslant n,1\leqslant i\leqslant m$。

★ 结果输出

将计算出的最少页面缺失总次数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3 11	8
4 5 6 7 6 4 5 5 3 2 1	

分析与解答：

1）算法描述

当内存访问请求要访问的页面 $\sigma(i)$ 不在高速缓存中时，采用如下贪心策略来确定高速缓存中与 $\sigma(i)$ 交换的页面。设此时高速缓存中的页面为 y_j，它下一次在 σ 中出现的位置为 next_j，$1\leqslant j\leqslant k$。取高速缓存 r 使得 $\text{next}_r=\max\limits_{1\leqslant j\leqslant k}\{\text{next}_j\}$，并交换页面 $\sigma(i)$ 与 y_r。

上述贪心算法可以具体实现如下。

算法中用数组 q 记录访问请求序列 σ，由于初始时页面 $1,2,\cdots,k$ 在高速缓存中，将 $\sigma(j)$ 存储在 $q[k+j]$ 中；数组 y 记录高速缓存中页面在 σ 中出现的位置；数组 s 记录页面位置，当 $s[j]=0$ 时页面 j 在低速内存中，当 $s[j]=r>0$ 时页面 j 在高速缓存 r 中；数组 next 记录 σ 中同一页面下一次访问请求的位置。

当内存访问请求要访问的页面 $\sigma(j)$ 时，由 request(j)响应访问请求。

```
public static int request(int j)
{
    if (fault(j)) { discard(j); return 1;}
    else { access(j); return 0; }
}
```

其中，fault(j)判断页面 $\sigma(j)$ 是否在高速缓存中。

```
public static boolean fault(int j)
{
    return s[q[k+j]]==0;
}
```

如果页面$\sigma(j)$在高速缓存中,则由 access(j)响应访问。

```
public static void access(int j)
{
    y[s[q[k+j]]]=k+j;
}
```

如果页面$\sigma(j)$不在高速缓存中,则由 discard(j)响应访问。

```
public static void discard(int j)
{
    int i=lfd();
    s[q[y[i]]]=0; s[q[k+j]]=i; y[i]=k+j;
}
```

先按贪心策略,由算法 lfd 确定高速缓存 i 使得 $next_i = \max\limits_{1\leq j\leq k}\{next_j\}$,然后实施页面调换。算法 lfd 根据当前高速缓存中页面计算要调出高速缓存的页面。

```
public static int lfd()
{
    int j=1;
    for(int i=2,tmp=next[y[1]];i<=k;i++)
        if (next[y[i]] > tmp) { tmp=next[y[i]]; j=i; }
    return j;
}
```

在响应访问前还需要进行一些预处理,由 prepro()完成。

```
public static void prepro()
{
    y=new int[k+1];
    s=new int[n+1];
    next=new int[k+m+1];
    for (int i=1; i<=n; i++) s[i]=0;
    for (int i=1; i<=k+m; i++) next[i]=m+k+1;
    for (int i=1; i<=k+m; i++)
    {
        next[s[q[i]]]=i;
        s[q[i]]=i;
    }
    for (int i=1; i<=k; i++) s[i]=y[i]=i;
    for (int i=k+1; i<=n; i++) s[i]=0;
}
```

2）算法正确性

按题意内存访问请求序列为 $\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$。设高速缓存中的页面为 y_j，它下一次在 σ 中出现的位置为 next_j，$1\leqslant j\leqslant k$。当内存访问请求要访问的页面 $\sigma(i)$ 不在高速缓存中时，算法 lfd 的贪心策略是选取高速缓存 r 使得 $\text{next}_r=\max\limits_{1\leqslant j\leqslant k}\{\text{next}_j\}$，并交换页面 $\sigma(i)$ 与 y_r。下面证明按此贪心策略，最优页调度问题具有贪心选择性质。

设算法 OPT 是页调度问题的一个最优算法。内存访问请求序列 σ 在 $\sigma(i)$ 处第 1 次发生页面缺失。算法 lfd 选择高速缓存中的页面 q 并交换页面 $\sigma(i)$ 与 q。下面要证明这个贪心选择是正确的，即存在一个最优页调度序列，在响应内存访问请求 $\sigma(i)$ 时，与算法 lfd 的选择相同。事实上，如果算法 OPT 产生的最优页调度序列在响应内存访问请求 $\sigma(i)$ 时选择高速缓存中的页面 p，且 $p\neq q$。设算法 OPT 在响应内存访问请求 $\sigma(t)$ 时，首次调出页面 q。设 $\sigma(a)$ 是对页面 p 的下一次访问请求，$\sigma(b)$ 是对页面 q 的下一次访问请求。由算法 lfd 的贪心策略可知，$a<b$，即内存访问请求序列 σ 为：$\sigma=\sigma(1),\cdots,\sigma(i),\cdots,\sigma(a)=p,\cdots,\sigma(b)=q,\cdots,\sigma(m)$。

现在构造一个新算法 OPT*，在响应内存访问请求 $\sigma(i)$ 时与算法 lfd 的选择相同，即选择高速缓存中的页面 q 并交换页面 $\sigma(i)$ 与 q。此后，算法 OPT* 与算法 OPT 在高速缓存中恰有 $k-1$ 个页面相同，而且在接下来的调度序列中算法 OPT* 与算法 OPT 在高速缓存中始终保持恰有 $k-1$ 个页面相同，直至两个调度序列完全相同。下面分两种情形讨论新算法 OPT*。

（1）情形 1：$i<t<b$。

考查内存访问请求 $\sigma(l)$，$i<l<t$。由于算法 OPT* 与算法 OPT 在高速缓存中恰有 $k-1$ 个页面相同，故算法 OPT* 的高速缓存中存在唯一页面 e 不在算法 OPT 的高速缓存中。

当 $\sigma(l)\neq e$ 时，算法 OPT* 的选择与算法 OPT 相同，而且算法 OPT* 与算法 OPT 在高速缓存中仍然保持恰有 $k-1$ 个页面相同。

当 $\sigma(l)=e$ 时，算法 OPT 发生页面缺失，但算法 OPT* 没有发生页面缺失。此时算法 OPT 用某页面替换 e，此后算法 OPT* 与算法 OPT 在高速缓存中仍然保持恰有 $k-1$ 个页面相同。

对于内存访问请求 $\sigma(t)$，既不在算法 OPT 的高速缓存中，也不在算法 OPT* 的高速缓存中。此时算法 OPT 选择 q，而算法 OPT* 选择 e，此后两算法的调度序列完全相同。

按此方式构造的算法 OPT* 的页面缺失次数不超过算法 OPT 的页面缺失次数。

（2）情形 2：$t\geqslant b$。

考查内存访问请求 $\sigma(l)$。当 $i<l<b$ 时，与情形 1 相同。通过类似分析可知，直至内存访问请求 $\sigma(b)=q$，算法 OPT* 与算法 OPT 在高速缓存中恰有 $k-1$ 个页面相同。注意到 $a<b$，前面在 $\sigma(l)=e$ 时的情况至少发生 1 次，故在内存访问请求 $\sigma(b)=q$ 之前，算法 OPT* 的页面缺失次数少于算法 OPT 的页面缺失次数。

对于内存访问请求 $\sigma(b)$，算法 OPT* 发生页面缺失，但算法 OPT 没有发生页面缺失。此时算法 OPT* 选择 e，此后两算法的调度序列完全相同。

由此可见，在此情形构造的算法 OPT* 的页面缺失次数也不超过算法 OPT 的页面缺失次数。

综上所述，算法 OPT* 构造的页面调度序列是满足贪心策略的最优调度序列。也就是

说,最优页调度问题具有贪心选择性质。

3) 算法计算复杂性

对每次页面缺失算法lfd需要$O(k)$时间选取高速缓存。因此,整个算法的计算时间为$O(km)$。算法需要的空间显然为$O(k+n+m)$。

算法实现题11-2 在线LRU页调度

★ 问题描述

页调度问题是系统软件设计中提出的一个基本问题。系统软件在进行内存管理时,将内存按其存取速度分成2级,即高速缓存和低速内存。内存被分成固定大小的页面进行管理。高速缓存可容纳k个页面,其他页面在低速内存中。页调度问题的输入是内存访问请求序列$\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$。当内存访问请求要访问的页面$\sigma(i)$在高速缓存中时,不需页面调度;而当页面$\sigma(i)$不在高速缓存中时,发生页面缺失,调度算法要确定高速缓存中与$\sigma(i)$交换的页面。页调度算法对于内存访问请求序列$\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$的耗费是算法在执行过程中产生的页面缺失次数。内存访问请求是无法预知的,它是随着时间的推移一个接着一个地给出的。

在线LRU(least recently used)算法:内存访问请求$\sigma(i)$发生页面缺失时,将高速缓存中最近访问时间最早的页面与$\sigma(i)$交换。

★ 算法设计

对于给定的2级内存页面分布以及内存访问请求序列$\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$,按照LRU算法,响应每一个内存访问请求。

★ 数据输入

由文件input.txt给出输入数据。第1行有3个正整数n,k和m,表示有n个内存页面$1,2,\cdots,n$。高速缓存可容纳k个页面。初始时页面$1,2,\cdots,k$在高速缓存中。内存访问请求序列的长度为m。第2行有m个正整数表示内存访问请求序列$\sigma=\sigma(1),\sigma(2),\cdots,\sigma(m)$,$1\leqslant\sigma(i)\leqslant n$,$1\leqslant i\leqslant m$。

★ 结果输出

将LRU算法的页面缺失总次数输出到文件output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3 11	9
4 5 6 7 6 4 5 5 3 2 1	

分析与解答:

算法实现时用数组q记录访问请求序列σ。由于初始时页面$1,2,\cdots,k$在高速缓存中,将$\sigma(j)$存储在$q[k+j]$中;数组y记录高速缓存中页面在σ中出现的位置;数组s记录页面位置,当$s[j]=0$时页面j在低速内存中,当$s[j]=r>0$时页面j在高速缓存r中。

当内存访问请求要访问的页面$\sigma(j)$时,由request(j)响应访问请求。

```
public static int request(int j)
{
    if (fault(j)) { discard(j); return 1;}
```

```
        else { access(j); return 0; }
    }
```

其中，fault(j)判断页面 $\sigma(j)$是否在高速缓存中。

```
    public static boolean fault(int j)
    {
        return s[q[k+j]]==0;
    }
```

如果页面 $\sigma(j)$在高速缓存中，则由 access(j)响应访问。

```
    public static void access(int j)
    {
        y[s[q[k+j]]]=k+j;
    }
```

如果页面 $\sigma(j)$不在高速缓存中，则由 discard(j)响应访问。

```
    public static void discard(int j)
    {
        int i=lru();
        s[q[y[i]]]=0; s[q[k+j]]=i; y[i]=k+j;
    }
```

先按 LRU 贪心策略，由算法 lru 确定高速缓存中最近访问时间最早的页面，然后实施页面调换。算法 lru 根据当前高速缓存中页面计算最近访问时间最早的页面。

```
    public static int lru()
    {
        int j=1;
        for(int i=2,tmp=y[1];i<=k;i++)
            if (y[i]<tmp) { tmp=y[i]; j=i; }
        return j;
    }
```

在响应访问前还需要做一些预处理，由 prepro()完成。

```
    public static void prepro()
    {
        y=new int[k+1];
        s=new int[n+1];
        for (int i=1; i<=k; i++) s[i]=y[i]=i;
        for (int i=k+1; i<=n; i++) s[i]=0;
    }
```

算法实现题 11-3　*k* 服务问题

★ 问题描述

假设平面上有 k 辆服务车位于 $p_1, p_2, \cdots, p_k$。对这 k 个服务车的服务请求序列的位置

是 $q_1, q_2, \cdots, q_n$。当前 k 个服务要按服务请求序列提出请求的先后次序响应每个服务。对服务请求 q_i 的响应就是从当前的 k 辆服务车中选取一辆服务车 j，从 j 的当前位置移动到服务请求 q_i 的位置。响应服务请求 q_i 的耗费是服务车 j 移动的距离。服务请求是在服务过程中一个接着一个地给出的。问如何调度最节省，即 k 辆服务车在服务过程中移动的总距离最短。

★ 算法设计

对于给定的 k 辆服务车以及服务请求序列 $q_1, q_2, \cdots, q_n$，设计一个最优调度算法，满足所有服务请求并使 k 辆服务车在服务过程中移动的总距离最短。

★ 数据输入

由文件 input.txt 给出输入数据。第1行有两个正整数 k 和 n，表示有 k 辆服务车和 n 个服务请求。接下来的 k 行中每行有两个数 x 和 y，表示服务车初始位置的 x 坐标和 y 坐标。紧接着的 k 行中每行也有两个数 x 和 y，表示服务请求位置的 x 坐标和 y 坐标。

★ 结果输出

将计算出的 k 辆服务车移动的最短总距离输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 5	3
1 0	
3 0	
0 0	
1 0	
0 0	
1 0	
0 0	
1 0	
0 0	
1 0	
0 0	
1 0	

分析与解答：

1) 动态规划算法

不失一般性可设 $p_1, p_2, \cdots, p_k$ 与 $q_1, q_2, \cdots, q_n$ 均互不相同。实际上如果上述 $n+k$ 个点中有相同的点 p 和 q，也不妨将它们看作不同的点，只不过此时它们之间的距离 $d(p, q)=0$。

设 $S=\{p_1, p_2, \cdots, p_k, q_1, q_2, \cdots, q_n\}$，$|S|=k+n=m$。

点集 S 的恰含 k 个点的子集共有 $a=\binom{m}{k}$ 个，分别记为 S_i，$1 \leqslant i \leqslant a$。

特别地，$S_1=\{p_1, p_2, \cdots, p_k\}$。

一般情况下的 k 服务问题可以表示为从点集 S 的初始状态 S_1 出发，经过一系列状态变化，不断响应 $q_1,q_2,\cdots,q_n$ 的服务请求，最终到达 S 的某个确定的状态 S_j。

由于集合 S 中的点是距离空间中的点，因此 S 中点的距离满足三角不等式性质。据此可推知，对于 k 服务问题的任何一个算法 A，一定可以找到一个懒算法 B，使 B 的耗费不超过 A。上面所说的懒算法 B 是指对于每个服务请求，算法 B 只移动 1 次。事实上，如果服务请求的位置是 p，算法 A 将服务从位置 q 移动到 r，再移动到 p。由三角不等式性质可知 $d(q,r)+d(r,p)\geqslant d(q,p)$。也就是说，这种移动不会好于直接将服务移动到 p。因此，下面只要讨论 k 服务问题的懒算法就足够了。

设从初始状态 S_1 出发，经过一系列懒移动来响应服务请求序列 σ，最终到达状态 S_j 的最小费用为 $c(\sigma,S_j)$。$c(\sigma,S_j)$ 满足如下动态规划递归式

$$c(\varepsilon,S_j)=\begin{cases}0 & j=1\\ \infty & j>1\end{cases}$$

$$c(\sigma v,S_j)=\begin{cases}\min\{c(\sigma,S_i)+d(S_i,S_j)\mid v\notin S_i,\ |S_i\cap S_j|=k-1\} & v\in S_j\\ \infty & v\notin S_j\end{cases}$$

其中，ε 表示空序列；σv 表示序列 σ 的下一服务请求是 v。

设 $\sigma_0=\varepsilon,\sigma_i=q_1,q_2,\cdots,q_i,1\leqslant i\leqslant n$。用数组 c 来记录 $c(\sigma_i,S_j)$ 的值，即 $c[i][j]=c(\sigma_i,S_j),0\leqslant i\leqslant n,1\leqslant j\leqslant a$。

根据上述递归式，从数组 c 的第 i 行可以在 $O(a^2)$ 时间内计算出第 $i+1$ 行的值。计算结束后，$\min\{c[n][j]\mid 1\leqslant j\leqslant a\}$ 即为所求的最小耗费。

上述算法所需的计算时间为 $O(na^2)=O\left(n\binom{n+k}{k}^2\right)$。

2）最小费用流算法

在一般情况下可将 k 服务问题变换为一个最小费用流问题。

设 k 个服务为 $s_1,s_2,\cdots,s_k$，服务请求序列为 $r_1,r_2,\cdots,r_n$。构造一个有 $2n+k+2$ 个顶点的网络 $G=(V,E)$ 如下。$V=\{s,s_1,s_2,\cdots,s_k,r_1,r_1',r_2,r_2',\cdots,r_n,r_n',t\}$。其中，$s$ 和 t 为分别为源顶点和汇顶点。网络 G 中每条边的容量均为 1。

从源 s 到每个 s_i 有一条费用为 0 的边，$1\leqslant i\leqslant k$；从每个 s_i 到汇 t 到有一条费用为 0 的边，$1\leqslant i\leqslant k$；从每个顶点 r_i' 到汇 t 到有一条费用为 0 的边，$1\leqslant i\leqslant n$。

从每个顶点 s_i 到每个顶点 r_j 有一条费用为 $d(s_i,r_j)$ 的边，$1\leqslant i\leqslant k,1\leqslant j\leqslant n$。

当 $i<j$ 时从顶点 r_i' 到顶点 r_j 有一条费用为 $d(r_i,r_j)$ 的边，$1\leqslant i<j\leqslant n$。

从每个顶点 r_i 到顶点 r_i' 有一条费用为 $-kk$ 的边，$1\leqslant i\leqslant n$。其中，kk 是一个很大的实数。

根据问题的输入构造网络 G 如下：

```
public static void main(String [ ] args)
{
    ReadStreams keyboard = new ReadStreams();
    k = keyboard.readInt();
    n = keyboard.readInt();
```

```
    s=0; t=k+2 * n+1; f=k;
    GRAPH G=new GRAPH(t+1);

    // 读入 k 个服务的初始位置
    for(int i=1;i<=k;i++){
        server[i][0]=keyboard.readInt();
        server[i][1]=keyboard.readInt();
    }

    // 读入 n 个服务请求的位置
    for(int i=1;i<=n;i++){
        r[i][0]=keyboard.readInt();
        r[i][1]=keyboard.readInt();
    }

    // 从源 s 到每个服务的边;从每个服务到汇 t 的边
    for (int j=1; j<=k; j++){
        G.insert(new EDGE(0, j, 1, 0));
        G.insert(new EDGE(j, t, 1, 0));
    }

    // 从 r 到 r'的边;从 r'到汇 t 的边
    for (int j=1; j<=n; j++){
        G.insert(new EDGE(k+2 * j-1, k+2 * j, 1, -kk));
        G.insert(new EDGE(k+2 * j, t, 1, 0));
    }

    // 从 s 到 r 的边
    for (int i=1; i<=k; i++)
        for (int j=1; j<=n; j++){
            double d=dist(server[i][0], server[i][1], r[j][0], r[j][1]);
            G.insert(new EDGE(i, k+2 * j-1, 1, d));
        }

    // 从 ri到 rj的边
    for (int i=1; i<=n; i++)
        for (int j=i+1; j<=n; j++){
            double d=dist(r[i][0], r[i][1], r[j][0], r[j][1]);
            G.insert(new EDGE(k+2 * i, k+2 * j-1, 1, d));
        }

    // 从源 s 到汇 t 的最小费用流
    minCost(G, s, t, f);
}
```

minCost(G, s, t, f)计算出网络 G 从源 s 到汇 t 的最小费用最大流 f。容易看出，网络 G 的最大流值为 k。由于网络 G 是一个有向无环图(DAG)，且每条边的容量均为整数 1，因此，可以用最小费用增广路算法，在 $O(kn^2)$时间内计算出网络 G 的最小费用最大流。网络 G 从源 s 到汇 t 的最小费用最大流可以分解为 k 条边不交 s-t 路。经过顶点 s_i 的 s-t 路上的顶点表示第 i 个服务响应的服务请求序列。由于 kk 是一个很大的实数，故费用 $-kk$ 使最小费用流必经过边(r_i, r_i')，$1 \leqslant i \leqslant n$，因此，找到的最小费用流对应于 k 服务问题的一个最优解。

参考文献

[1] 王晓东. 算法设计与分析[M]. 4版. 北京：清华大学出版社，2018.

[2] 王晓东，傅清祥，叶东毅. 算法与数据结构学习指导与习题解析[M]. 北京：电子工业出版社，2000.

[3] Aho A V，Hopcroft J E，Ullman J D. The Design and Analysis of Computer Algorithms[M]. New York：Addison-Wesley，1974.



普通高等教育“十一五”国家级规划教材
21世纪大学本科计算机专业系列教材

近期出版书目

- 计算概论(第2版)
- 计算概论——程序设计阅读题解
- 计算机导论(第3版)
- 计算机导论教学指导与习题解答
- 计算机伦理学
- 程序设计导引及在线实践(第2版)
- 程序设计基础(C语言)(第2版)
- 程序设计基础(C语言)实验指导(第2版)
- 离散数学(第3版)
- 离散数学习题解答与学习指导(第3版)
- 数据结构(STL框架)
- 算法设计与分析(第4版)
- 算法设计与分析习题解答(第4版)
- C++程序设计(第3版)
- Java程序设计(第2版)
- 面向对象程序设计(第3版)
- 形式语言与自动机理论(第3版)
- 形式语言与自动机理论教学参考书(第3版)
- 数字电子技术基础
- 数字逻辑
- FPGA数字逻辑设计
- 计算机组成原理(第3版)
- 计算机组成原理教师用书(第3版)
- 计算机组成原理学习指导与习题解析(第3版)
- 微机原理与接口技术
- 微型计算机系统与接口(第2版)
- 计算机组成与系统结构(第2版)
- 计算机组成与体系结构习题解答与教学指导(第2版)
- 计算机组成与体系结构(第2版)
- 计算机系统结构教程
- 计算机系统结构学习指导与题解
- 计算机系统结构实践教程
- 计算机操作系统(第2版)
- 计算机操作系统学习指导与习题解答
- 编译原理
- 软件工程(第3版)
- 计算机图形学
- 计算机网络(第4版)
- 计算机网络教师用书(第4版)
- 计算机网络实验指导书(第3版)
- 计算机网络习题解析与同步练习(第2版)
- 计算机网络软件编程指导书(第2版)
- 人工智能
- 多媒体技术原理及应用(第2版)
- 计算机网络工程(第2版)
- 计算机网络工程实验教程
- 信息安全原理及应用